НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:

Завідувач кафедри

_____ Сергій, СТІРЕНКО

«___»_____2020 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Комп'ютерні системи та мережі»

спеціальності 123 «Комп'ютерна інженерія»

на тему: «Спосіб кластеризації мобільних SDN мереж»

Виконав:
студент IV курсу, групи ІО-64
Василенко Олександр Вікторович _____

Керівник:
Асистент
Калюжний Олександр Олегович _____

Консультант з нормоконтролю:
Професор кафедри ОТ, д.т.н.,
Сімоненко Валерій Павлович _____

Рецензент:
Доц. каф. СПіСКС к.т.н., доц.,
Орлова Марія Миколаївна _____

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2020 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 123 «Комп'ютерна інженерія»

Освітньо-професійна програма «Комп'ютерні системи та мережі»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____Сергій, СТІРЕНКО
«___»_____2020 р.

ЗАВДАННЯ

на дипломний проєкт студенту

Василенку Олександру Вікторовичу

1. Тема проєкту «Спосіб кластеризації мобільних SDN мереж», керівник проєкту Калюжний Олександр Олегович, асистент, затверджені наказом по університету від «07» травня 2020 р. № 1081-с

2. Термін подання студентом проєкту *26 травня 2020р.*

3. Вихідні дані до проєкту *див. технічне завдання*

4. Зміст пояснювальної записки *Аналіз і характеристика існуючих рішень об'єкта проектування, обґрунтування оптимального реалізації розроблюваного додатку, вибір інструментів та технологій і обґрунтування вибору, розробка додатку, основні рішення з реалізації. Висновки.*

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслеників, плакатів, презентацій тощо) *приниципова схема, функціональна схема, структурна схема*

6. Консультанти розділів проєкту

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
нормоконтроль	Сімоненко В. П., професор, д.т.н.		

7. Дата видачі завдання 01.09.2019

Календарний план

№ з/п	Назва етапів виконання дипломного проекту	Термін виконання етапів проекту	Примітка
1.	*Затвердження теми роботи*	*01.09.2019*	
2.	*Вивчення та аналіз завдання*	*15.12.2019-15.03.2020*	
3.	*Розробка архітектури додатку*	*15.03.2020-25.03.2020*	
4.	*Написання програмної частини*	*25.03.2020-05.04.2020*	
5.	*Тестування та виправлення помилок*	*05.04.2020-15.04.2020*	
6.	*Оформлення пояснювальної записки*	*15.04.2020-20.05.2020*	
7.	*Захист програмного продукту*	*25.04.2020*	
8.	*Передзахист*	*26.05.2020*	
9.	*Захист*	*18.06.2020*	

Студент Олександр ВАСИЛЕНКО

Керівник Олександр КАЛЮЖНИЙ

Анотація

В даному бакалаврському дипломному проекті було запропоновано алгоритм кластеризації мобільнлї мережі SDN. Було проведено огляд існуючих алгоритмів кластеризації мережі SDN, розроблено власний алгоритм кластеризації. Виконано програмну реалізацію розробленого алгоритму, протестовано її роботу. Виконано порівняння результатів моделювання алгоритму із іншими алгоритмами.

Abstract

In this bachelor's diploma project the mobile SDN network clustering algorithm has been proposed. A review of existing SDN clustering algorithms has been performed, the new clustering algorithm has been developed. A software implementation of the developed algorithm has been created and tested. The algorithm’s modelling results has been compared with those of the other algorithms.

ВІДОМІСТЬ ДИПЛОМНОГО ПРОЄКТУ

№ з/п	Формат	Позначення	Найменування	Кількість листів	Примітка
1	А4		Завдання на дипломний проект	2	
2	А4	ДП 6404. 02.000 ТЗ	Технічне Завдання	3	
3	А4	ДП 6404. 03.000 ПЗ	Пояснювальна записка	64	
4	А4	ДП 6404. 04.000 Д1	Принципова схема	1	
5	А4	ДП 6404. 05.000 Д2	Функціональна схема	1	
6	А4	ДП 6404. 06.000 Д3	Структурна схема	1	
7	А4	ДП 6404. 07.000 Д4	Лістинг програми	28	

Зм.	Арк.	№ документа	Підп.	Дата	ДП.6404.01.000 ВП		
Розробив		Василенко О.В.			Спосіб кластеризації мобільних SDN мереж Відомість дипломного проєкту	Літ.	Аркуш 1 / шв 1
Перевір.		Калюжний О.О.				Т	
Н.конт		Сімоненко В.П.				НТУУ «КПІ імені Ігоря Сікорського», ФІОТ Група ІО - 64	
Затв.							

Технічне завдання

до дипломного проєкту

на тему «Спосіб кластеризації мобільних SDN мереж»

ЗМІСТ

1.НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Найменування: «Спосіб кластеризації мобільних SDN мереж».

Область застосування: програма може бути використана для кластеризації мобільних SDN мереж.

2.ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки є завдання на виконання бакалаврського дипломного проекту, затверджене кафедрою обчислювальної техніки Національного технічного університету України «Київський політехнічний інститут ім. Ігоря Сікорського».

3.МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою розробки є забезпечення рівномірного навантаження контролерів у мережі SDN.

4. ДЖЕРЕЛА РОЗРОБКИ

Джерелом розробки є науково-технічна література, публікації в спеціалізованих періодичних виданнях, довідники по платформах дистанційного навчання, публікації в мережі Інтернет по даній темі.

5.ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до програмного продукту, що розробляється

Додаток, що розробляється повинен:

- Створювати, зберігати, редагувати граф мережі
- Моделювати обраний алгоритм кластеризації
- Мати можливість перегляду та аналізу результатів моделювання
- Мати можливість розширення можливостей за рахунок додавання інших аолгоритмів

5.2. Вимоги до пристрою

- Процесор з тактовою частотою не менше ніж 2 ГГц;
- Оперативна пам’ять об’ємом не менше ніж 1 Гб;
- Встановлена віртуальна машина JVM не нижче 8 версії

6. ЕТАПИ РОЗРОБКИ

	Дата
Вивчення необхідної літератури	19.02.2020
Складання і узгодження технічного завдання	06.03.2020
Написання вступної частини та огляд рішень	19.03.2020
Розробка архітектури додатку	03.04.2020
Написання програмної частини	10.04.2020
Тестування та виправлення помилок	01.05.2020
Оформлення документації дипломного проекту	15.05.2020
Попередній захист та проходження нормативного контролю	29.05.2020
Захист дипломного проекту	15.06.2020

Пояснювальна записка
до дипломного проєкту
на тему: «Спосіб кластеризації мобільних SDN мереж»

Київ – 2020 року

ЗМІСТ

СПИСОК СКОРОЧЕНЬ

SDN	Software Defined Network
API	Application Programming Interface
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
MPO	Метричне Розміщення Об’єктів
WAN	Wide Area Network
NFV	Network Functions Virtualization
DBCP	Data Base Connection Pool
OS	Operating System
JRE	Java Runtime Environment
SDK	Software Development Kit
JDK	Java Development Kit
JVM	Java Virtual Machine

ВСТУП

Суть концепції, що швидко розвивається, Software Defined Networking (SDN), що з'явилася в останні роки, полягає в відокремленні рівня управління від рівня даних, що спрощує управління мережею та підвищує гнучкість та масштабованість. Контролер на рівні управління керує мережевими маршрутизаторами, надаючи маршрутизаторам правила, які повинні виконувати перенесення пакетів. Це спрощує функції маршрутизаторів і прискорює їх роботу [2].

Успішне розгортання контролерів у великих мережах дозволяє максимально використати існуючу структуру з'єднань між мережевими вузлами. Ключовим фактором для мереж SDN є швидкий обмін даними та надійне з'єднання між маршрутизатором та контролером, який ним керує. Одного контролера недостатньо для управління всіма вузлами в налаштованій мережі за допомогою одного програмного забезпечення, оскільки продуктивність обмежена, а затримка поширення сигналу між контролером та всіма вузлами в мережі дуже велика; Крім того, один контролер в мережі не може відповідати вимогам щодо відмовок. Тому має сенс розміщувати декілька контролерів у різних точках мережі, які разом керують усім рівнем даних. Тому питання про те, як кластерувати мережу та де розміщувати контролери, було б найбільш ефективним [3].

При вирішенні цієї проблеми слід враховувати кілька факторів, і отримане рішення забезпечує ефективну та стабільну роботу мережі. Це, в першу чергу, затримка передачі сигналу між вузлами мережі та контролерами, які ними керують: вона повинна бути мінімізована. Наступне, що слід врахувати, - це навантаження контролера: зрозуміло, що якщо перевищити можливості контролера, ефективність знизиться. Іншими важливими факторами є відмовостійкість та затримка передачі сигналу між контролерами (у ситуації, коли логіка управління мережею розподіляється

між кількома контролерами, вони повинні взаємодіяти для синхронізації своїх позицій: забезпечуючи таким чином цілісність глобальної мережі) [4].

Оскільки розташування контролерів - це низка факторів, які слід враховувати при вирішенні проблеми, її часто визначають як задачу комбінаторної оптимізації. Ця проблема є повною NP [1]. Одним із підходів до пошуку оптимального рішення є використання евристичних алгоритмів, але їх недоліком є те, що для пошуку рішення у великих мережах потрібно дуже багато часу [2].

З іншого боку, для кожного вузла мережі можна обчислити певні показники, пов'язані з перерахованими вище факторами (затримка, баланс навантаження, допущена несправність) та оцінити, наскільки вдалою є ця точка для розміщення контролера. Цей підхід застосовується в ряді алгоритмів, розроблених в останні роки [5] [6] [7] [8] [9]. Звичайно, знайти ідеальний баланс непросто, і той чи інший алгоритм може дати кращі результати, залежно від характеристик конкретної мережі, відносної важливості різних факторів для її роботи.

Пропонується ще один алгоритм для вирішення проблеми розташування контролера.

РОЗДІЛ 1.

ОГЛЯД ІСНУЮЧИХ МЕТОДІВ КЛАСТЕРИЗАЦІЇ

1.1. Мережа SDN: основні поняття

Програмно-конфігурована мережа (SDN) - це відносно нова технологія програмування комп'ютерних мереж. В основі концепції SDN лежить розділення рівня управління та рівня даних у мережі. Правила маршрутизації даних у мережі визначаються централізованим пристроєм - програмованим контролером, тоді як пристрої передачі даних (вузли) в мережі керуються цими правилами і, таким чином, здатні швидше надсилати пакети [1]. Таке управління можливе завдяки програмному інтерфейсу між контролером та вузлами. Найпоширеніший інтерфейс - OpenFlow.

Архітектура мережі SDN охоплює 3 рівні (рис. 1.1) [2]:

- інфраструктурний рівень: мережеве обладнання (пристрої та канали передачі даних);

- рівень управління: мережева операційна система, яка забезпечує програмним інтерфейсом для управління мережею;

- прикладний рівень: віртуалізація мережі, програми захисту, управління та інші інструменти для підвищення ефективності управління мережею.

Основні ідеї та принципи концепції SDN наступні:

- розділення процесів передачі та управління даними;

- єдиний, незалежний від постачальника інтерфейс між шаром управління та шаром передачі даних;

- логічно централізоване управління мережею за допомогою контролера із встановленою мережевою операційною системою та встановленими мережевими додатками;

- віртуалізація мережевих фізичних ресурсів.

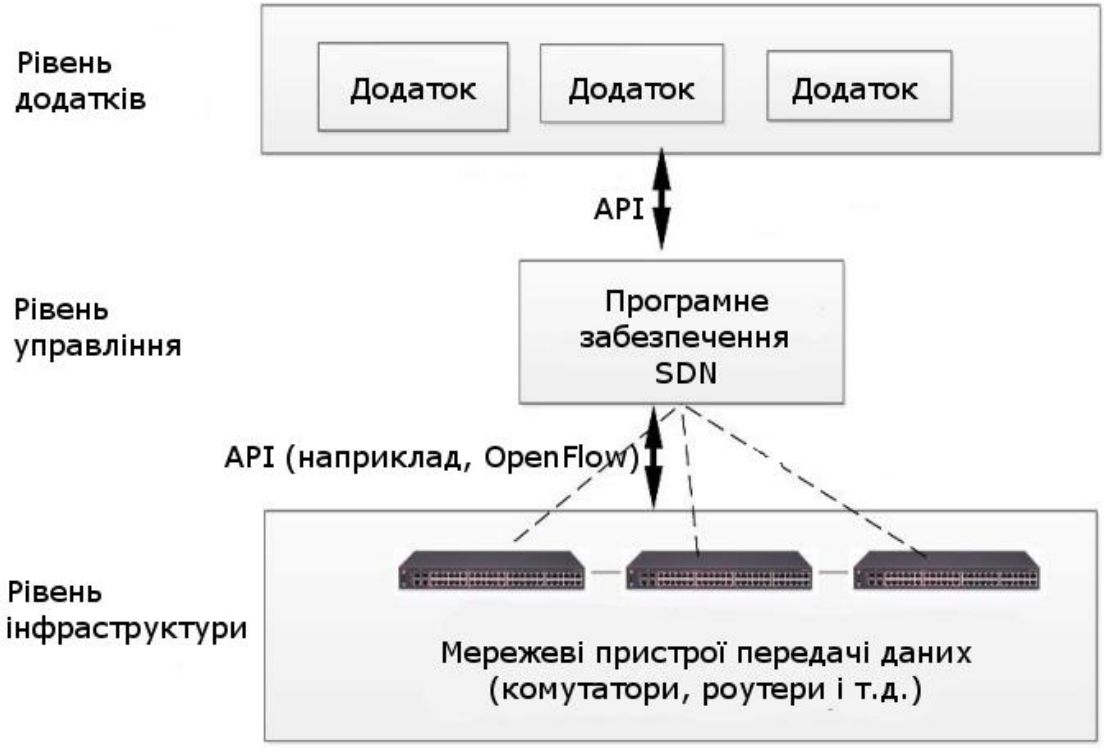


Рис. 1.1 Мережа SDN

Відокремлення функції управління на окремий шар та перенесення її до центрального компонента (мережевого контролера) спрямоване на оптимізацію конфігурації мережі для завдань додатків, і зокрема дозволяє отримати такі переваги: формати даних, правила обробки та технології передачі в маршрутизаторі які не обмежуються обладнанням або виробником; передача може бути визначена багатовимірним вектором, що охоплює області з різних рівнів моделі мережевої взаємодії; автоматизовані методи корекції потоку залежно від завантаження компонентів та інших критеріїв можуть застосовуватися при встановленні правил передачі пакетів; мережеві контролери можуть бути інтегровані в мережеві домени, що дозволяють оптимізувати та резервувати канали передачі.

Ідея SDN створити єдиний, незалежний від постачальника мережі, інтерфейс між контролером та мережевим транспортним середовищем відображена в протоколі OpenFlow (мал. 1.2), що дозволяє користувачам визначати та керувати, з ким, за яких умов та з чим, з якою якістю буде

взаємодія мережа. Протокол підтримує три типи повідомлень: контролер-маршрутизатор, асинхронне та симетричне.

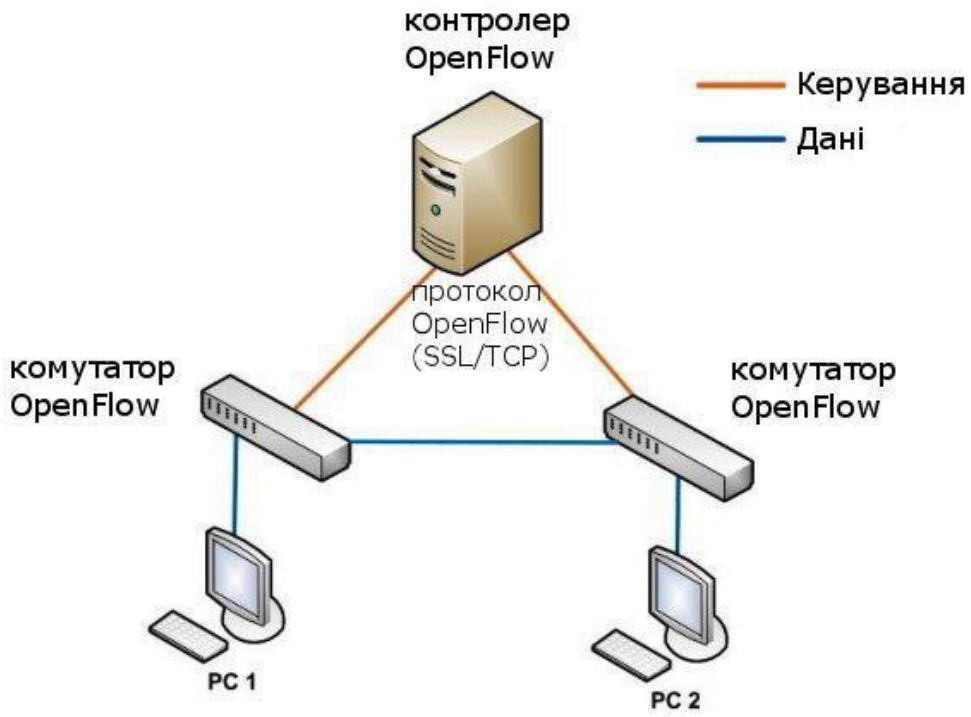


Рис. 1.2. OpenFlow архітектура мережі

Повідомлення між контролером та маршрутизатором ініціюються контролером і використовуються для безпосереднього управління та контролю стану маршрутизатора. Цей тип повідомлень може використовуватися контролером для встановлення параметрів конфігурації маршрутизатора, збору статистики, додавання, видалення та зміни записів у тематичних таблицях.

Асинхронні повідомлення, ініційовані маршрутизатором, повідомляють контролер про події в мережі (прихід пакету або видалення запису з таблиці коли час виходить) та зміни стану або помилки маршрутизатора.

Симетричні повідомлення можна запускати без потреби в запиті і використовуються для встановлення з'єднання, а також для контролю затримок з'єднання контролер-маршрутизатор, для вимірювання пропускної здатності або для перевірки активності з'єднання.

Однією з ідей SDN є віртуалізація мереж з метою більш ефективного використання мережевих ресурсів. Віртуалізація мережі - це ізоляція

мережевого трафіку - мультиплексування декількох потоків даних з різними характеристиками в межах однієї логічної мережі, яка може спільно використовувати одну фізичну мережу з іншими логічними мережами або фрагментами мережі. Кожен з цих фрагментів надає власні адреси, алгоритми маршрутизації, управління якістю обслуговування тощо, які може використовувати. Віртуалізація мережі дозволяє: підвищити ефективність розподілу мережевих ресурсів і балансувати навантаження; ізолювати потік різних користувачів та додатків у межах однієї фізичної мережі; використовувати політику маршрутизації та політику управління потоками даних адміністраторам різних фрагментів; проводити мережеві експерименти з використанням реальної фізичної мережевої інфраструктури; використовувати в кожному розділі лише послуги, необхідні для спеціальних програм.

Засоби віртуалізації SDN - це проксі-програми, які використовують протокол OpenFlow і працюють на рівні між маршрутизаторами та контролерами SDN. За допомогою цих програм ви можете створювати логічні сегменти, які забезпечують взаємну ізоляцію таких логічних мереж, використовуючи різні алгоритми управління потоком даних. Це означає, що кожен контролер керує лише власною логічною мережею і не може впливати на роботу іншого. Для контролера, який спілкується з пристроями OpenFlow за допомогою проксі-програми, повідомлення з'являється так, ніби контролер підключений до звичайної мережі SDN. Таким чином, віртуалізація не змінює спосіб роботи контролера.

Для простоти первісна розробка та реалізація OpenFlow передбачала єдиний контролер в мережі. Однак, оскільки масштаби мереж OpenFlow швидко зростали, стало зрозуміло, що покладатися на єдиний контролер для всієї мережі було неефективним з кількох причин.

По-перше, трафік у напрямку централізованого контролера збільшується із кількістю маршрутизаторів. По-друге, якщо мережа має великий діаметр, деякі вузли дуже довго чекатимуть інструкцій від

контролера, незалежно від місця розташування контролера. Нарешті, складність системи полягає в роботі контролера, створюючи таким чином чергу, яка може загальмувати всю мережу.

Розподілена модель SDN має на меті виправити проблему (що може призвести до несправності всієї мережі) та покращити розширення, розподіливши навантаження між кількома контролерами. Розподілені рівні управління SDN були розроблені таким чином, щоб вони були досить чутливими для управління локальними подіями в центрах обробки даних, де контролери обмінюються великою кількістю інформації для забезпечення глобальної та детальної цілісності мережі. Зокрема, розподілена архітектура SDN для багатодоменних SDN, в яких використовується все, починаючи від широкосмугової технології оптичного волокна до безпровідних ланок з обмеженою смугою пропускання, легко адаптується до потреб користувачів і додатків. Крім того, розподілений контролер є більш чутливим, надійним та здатним швидше та ефективніше керувати глобальними подіями.

Запропоновані розподілені рішення можна розділити на три типи. По-перше, це рішення, спрямовані на підвищення продуктивності окремих контролерів, таких як Maestro та McNettle, застосовуючи паралелізм на рівні маршрутизатора.

По-друге, архітектури із розподіленим контролером, такі як FlowVisor, Onix, HyperFlow, Devoflow. Поки підтримується логічно централізоване управління, рівні управління розподіляються між різними частинами мережі: використовуються розподілена синхронізація між локальними подіями, розподілені хеш-таблиці, для створення кластерних контролерів або навіть розподіленої файлової системи [2]. Приклад розподіленої архітектури показаний на рис. 1.3.

Третє рішення - багаторівнева архітектура з розподіленими контролерами. Наприклад, Kandoo визначає два ієрархічні рівні контролерів: нижній рівень контролерів, які не пов'язані один з одним, і кожен з яких управляє власною групою маршрутизаторів без інформації про глобальний

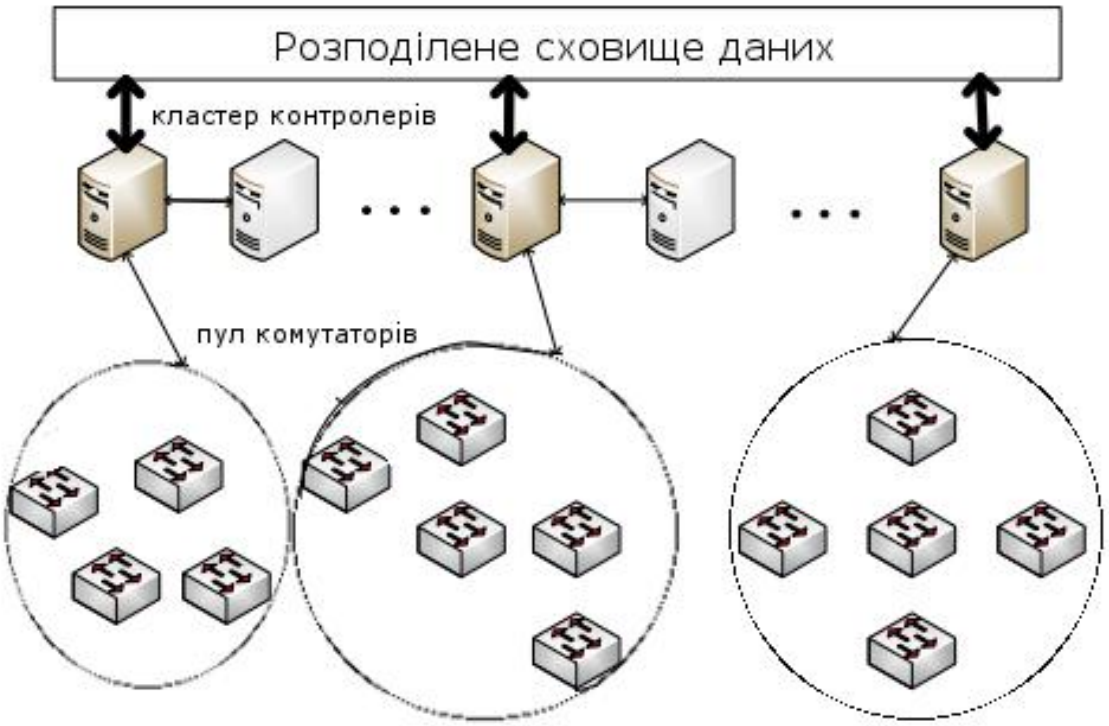


Рис. 1.3. Архітектура мережі із розподіленим контролером

мережі, і верхній рівень - логічно централізований кореневий контролер, який відповідає за стан мережі.

Наприклад, розглянемо архітектуру HyperFlow. Її завдання - спростити масштаб мережі, забезпечивши логічну централізацію управління мережею: всі контролери розглядають однаковий глобальний «вигляд» мережі та обслуговують запити локально без використання активних дзвінків. Таким чином, час налаштування потоку зводиться до мінімуму [10]. HyperFlow та подібні архітектури прозорі з точки зору стандарту OpenFlow, тобто не потребують змін.

Архітектура HyperFlow складається з ефективних маршрутизаторів OpenFlow, контролерів, кожен з яких виконує екземпляр HyperFlow, та систем розподілу подій для зв'язку між контролерами. Усі контролери бачать однаковий глобальний "вигляд" всієї мережі та діють так, ніби вони керують усією мережею. Усі вони мають однакову програму. Кожен маршрутизатор призначається певному контролеру. Якщо контролер виходить з ладу, маршрутизатори пошкодженого контролера повинні бути відновлені і

повинен бути призначений активний контролер поряд. Кожен контролер безпосередньо керує призначеними маршрутизаторами і впливає на роботу інших, спілкуючись з іншими контролерами. Огляд HyperFlow показаний на малюнку 1.4, де вгорі зображення зображена система взаємодій між контролерами [10].

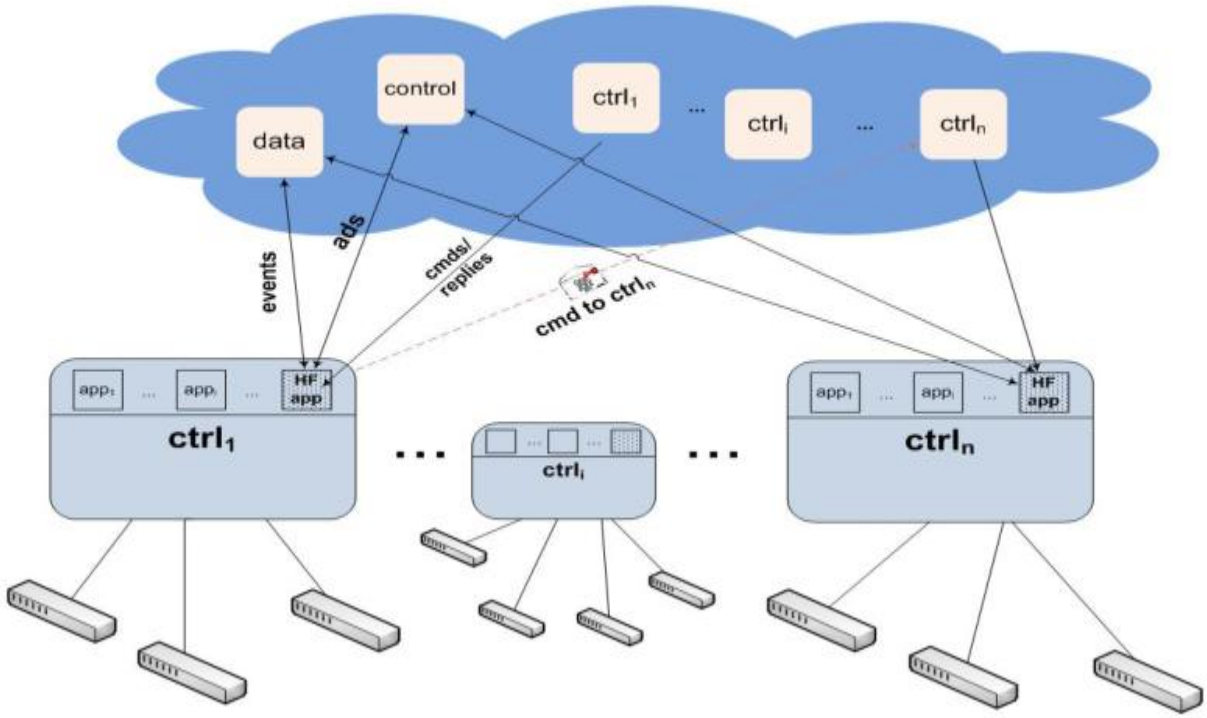


Рис. 1.4. Схема роботи HyperFlow

HyperFlow посилає події, які змінюють стан мережі в систему зв'язку публікація/підписка на кожному контролері для досягнення цілісності представлення мережі серед усіх контролерів [10]. Інші контролери відтворюють трансляцію подій на мережевих моделях для відновлення поточного стану мережі.

За допомогою видалення з маршрутизаторів функції управління SDN дозволяє цим пристроям перенаправляти всі ресурси для прискорення трафіку, що значно підвищує продуктивність. Віртуалізація управління мережею зменшує витрати на їх створення та утримання.

SDN дозволяють адміністраторам додавати нові функції до існуючої мережевої архітектури. Однак такі програмні рішення є універсальними,

оскільки вони не мають залежності від прошивки маршрутизатора кожного постачальника.

1.2. Завдання зі кластеризації мережі SDN

Оскільки SDN відокремлює рівень управління та рівень даних, створюється питання розташування контролерів. Мережа повинна бути розділеною і один контролер повинен бути розміщений у кожній підмережі.

Ключовим фактором тоді є показник затримки мережі. Завдання з оптимізації існують у багатьох контекстах, у тому числі за межами світу комп'ютерних мереж: у більш загальному випадку вони відомі як завдання розміщення об'єктів, і вирішенні проблеми полягає у забезпеченні найшвидших відповідей, особливо для розташування пожежних станцій, складів біля заводів, оптимізації розташування проксі-серверів.

Проблема розміщення об'єкта - це проблема Вебера, розміщення об'єкта для мінімізації вимірюваної суми відстаней до певної сукупності точки. Більш складні завдання цієї дисципліни виникають, коли існують обмеження щодо розміщення об'єктів та використання більш складних критеріїв оптимізації.

У базовому формулюванні завдання розміщення об'єктів складається з потенційних точок розміщення, що складаються з L точок, які можуть відкривати об'єкти, і D точок, які необхідно розмістити. Мета полягає у виборі меншої частини F точок для розташування об'єктів для мінімізації величини відстаней від кожної точки обслуговування до найближчого місця об'єкта, а також суми витрат на розміщення об'єкта.

Оптимальне рішення задачі розміщення таблиць у загальних графах характеризується як NP-комплексне і може бути отримане шляхом його зменшення у випадку кратності (наприклад). Було розроблено кілька алгоритмів для завдань розміщення об'єктів у великій кількості варіацій.

Без припущень щодо властивостей відстаней між клієнтами та місцями розташування (зокрема, не припускаючи, що відстань задовольняє нерівності

трикутника), проблема відома як неметрична задача розміщення об'єкта і може бути апроксимована фактором O (log n) [8].

Якщо припустити, що відстані між клієнтами та місцями розташування не орієнтовані і трикутник нерівний, то мова йде про проблему метричного розміщення об'єктів (MPO).

Завдання щодо мінімального розміщення об'єктів шукає розміщення, яке мінімізує відстані від точки до найближчого місця розташування. Офіційне визначення таке: Якщо задано безліч точок P ⊂ ℝd, потрібно знайти безліч точок S ⊂ ℝd, | S | = K, таке, що значення max p ∈ P (min q ∈ S (d (p, q))) буде мінімальним [8].

Усі такі завдання є NP-повними, і всі вони мають вагомі типи, які вирішують задачі з урахуванням різного значення вузлів.

У задачі компонування контролера є кілька взаємопов'язаних завдань: визначення оптимальної кількості контролерів у мережі, розділення мережі на кластери, вибір місця розташування контролера в кожній підмережі за результатом кластеризації. Приклад кластеризації мережі та розташування контролерів у ній показаний на рис. 1.5, де пунктирна лінія вказує розділення мережі на підмережі.

Рішення проблеми компонування контролерів впливає на всі аспекти рівня управління в програмно-конфігурованій мережі, від можливостей поширення стану до допущення помилок та продуктивності. Рішення визначає доступність та терміни обранння вузлів мережі в WAN з високою затримкою.

Це має практичне значення для розробки програмного забезпечення, оскільки впливає на те, чи можуть контролери реагувати на події в режимі реального часу та чи передають вони інструкції маршрутизаторам заздалегідь.

У більшості документів ця проблема визначається як проблема багатоцільової комбінаторної оптимізації, є NP-повною, а для пошуку оптимального рішення використовуються евристичні алгоритми [5]. Однак ці

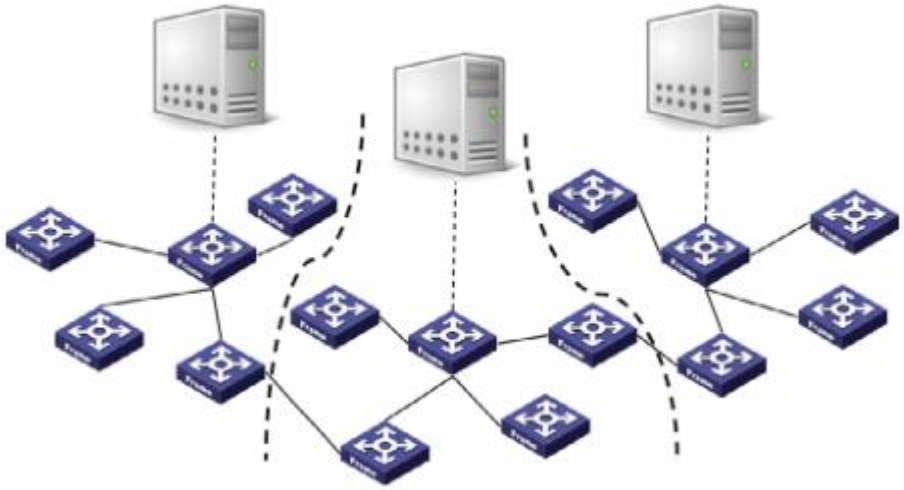


Рис. 1.5. Приклад кластеризації мережі

евристичні алгоритми мають недоліки "дотримуватися" локальних оптимальних рішень та знаходити довго рішення [6].

З іншого боку, існують підходи до розміщення контролерів у мережевих точках, які відповідають визначеним критеріям вибору. Деякі з цих методів обговорюються в наступних розділах.

Перші пропозиції щодо вирішення проблеми розгортання контролера полягали в тому, щоб спробувати зосередити увагу на затримці передачі сигналу в мережі, незалежно від навантаження на контролери, але це ключове питання для реальних мереж. Тому було запропоновано модифіковану постановку задачі: проблема розміщення контролера ємності [9].

Хоча головна мета успішного розміщення контролерів - мінімізувати затримки сигналу між вузлами мережі та контролерами, не тільки затримки враховуються. Розміщення контролерів також повинно відповідати певним вимогам щодо надійності та цілісності. Наприклад, розглянемо оптимальне регулювання k = 5 контролерів щодо максимальної затримки в графі мережі, показаному на рисунку. 1.3-1.6 та аналізуємо проблеми надійності, які можуть виникати в мережі.

1.2.1. Збої контролерів

Збільшення кількості контролерів, розподілених по мережі певно рівномірно, зменшує максимальну затримку між вузлами та контролерами в мережі [5]. Це також збільшує відмовостійкість при закритті контролерів. У разі відмови контролера всі вузли, якими він керує, можуть бути делеговані іншим контролерам, тобто у випадку кожного конкретного вузла, другому, найближчому до цього вузла контролеру. Це робиться за допомогою таблиці призначення або звичайного алгоритму маршрутизації, щоб знайти найкоротший маршрут. Таким чином, якщо є хоча б один контролер, всі вузли можуть продовжувати працювати. Однак затримки з вузлів до нових контролерів можуть значно збільшитися порівняно з початковою ситуацією. На малюнку 1.6 вказано збій чотирьох із п’яти контролерів у мережі (припинені контролери позначені червоними хрестиками та один контролер, що працює колом). Затримки від вузлів до останнього функціонуючого контролера забарвлені: зелений колір означає затримку рівну нулю, жовтий колір означає затримку 50% від діаметра сітки, а червоний - затримку 100% діаметра сітки. Найгірший сценарій цієї відмови контролерів полягає в тому, що контролер, що залишився, знаходиться найдалі від центру мережі. Це означає, що запити від деяких вузлів повинні пройти майже через всю мережу, щоб дістатися до контролера. Для підвищення стійкості мережі до цього явища розташування контролерів повинно враховувати не тільки затримку, досягнуту без відмов мережі, але і найгіршу затримку відмов контролерів.

1.2.2. Порушення цілісності мережі

На відміну від відмов контролера, видалення фізичних частин мережі, таких як вузли та з'єднання, має більший вплив на стабільність мережі, оскільки це змінює саму топологію. Найкоротші шляхи між деякими вузлами змінюються, викликаючи зміну затримок та, можливо, перепризначення деяких вузлів іншим контролерам. Крім того, серйозна проблема полягає в

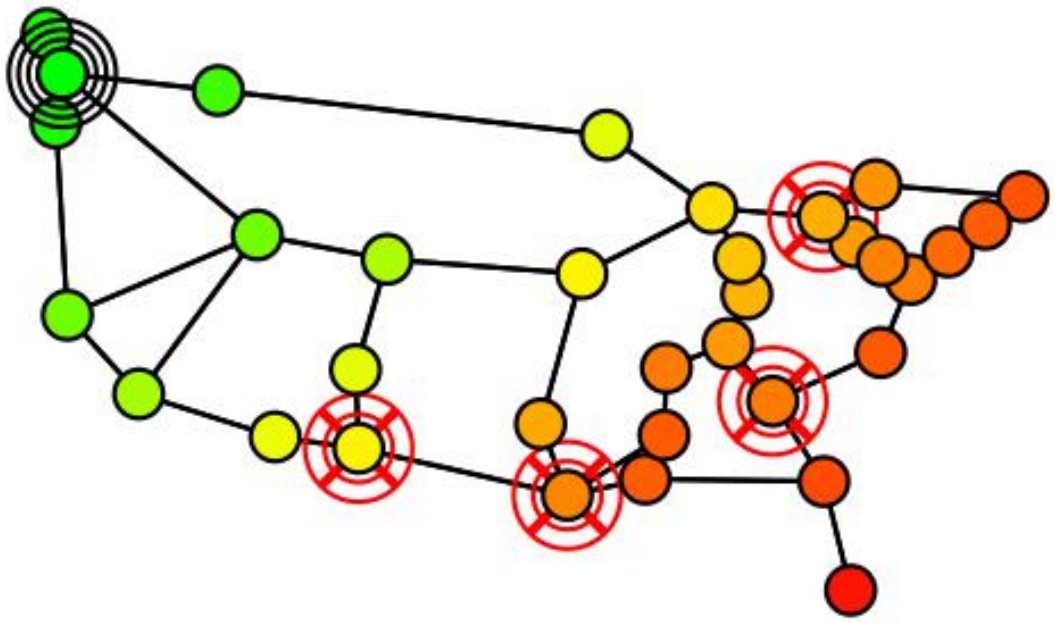


Рис. 1.6. Випадок коли деякі мережеві контролери виходять з ладу

тому, що всі частини мережі мають ризик переривання (відключення від решти мережі) через збій вузла або комутатора. У гіршому випадку деякі вузли більше не будуть підключені до жодного контролера, оскільки всі контролери не можуть отримати доступ до них. Такі вузли все ще активні і здатні передавати дані, але більше не можуть звертатися до контролера за інструкціями.

На малюнку 1.7 показаний найгірший сценарій. Частини, залишені без нагляду, позначаються знаками запитання.

Тому для всієї підмережі, що складається з позбавлених вузлів від контролера, будь-яка функція, застосована в контролері, недоступна, навіть якщо самі вузли функціональні та взаємопов'язані.

1.2.3. Нерівномірне навантаження на контролери

Скажімо, ми використовуємо один покажчик затримки, який є найкоротшим шляхом між вузлом та контролером, щоб вибрати контролер, який керує вузлом. Розділ мережі на підмережі, кожна з яких управляється відповідним контролером, показаний на малюнку. 1.8. Кількість вузлів, присвоєних кожному контролеру, не є однаковою: від 4 до 10 вузлів на

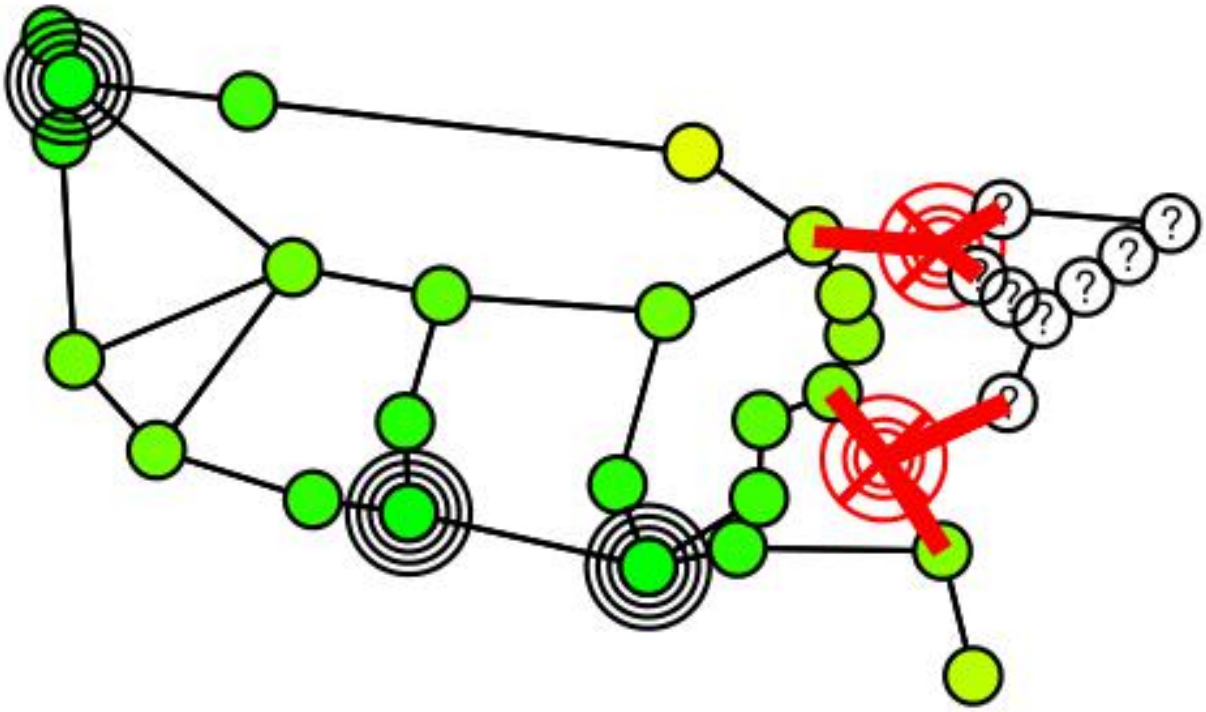


Рис. 1.7. Поломка деяких вузлів та з'єднань у мережі

контролер. Чим більше вузлів, контрольованих контролером, тим більше навантаження на контролер. Це особливо важливо під час частого доступу вузлами до частих контролерів, особливо за допомогою віртуалізації мережевих функцій (NFV). Зі збільшенням кількості запитів, які приймає контролер від вузла, зростає й ризик додаткових затримок через збір запитів з боку контролера.

Призначення вузлів контролерам повинно бути збалансованим, щоб протистояти перевантаженню контролерів. Тому важливим і актуальним є вирішення проблеми розміщення контролерів, що демонструє комплексний підхід і особливо враховує аспект балансу навантаження.

1.2.4. Затримка між контролерами

Зрозуміло, що ситуація, коли в мережі є контролер, не може відповідати вимогам надійності. Але коли є кілька контролерів, виникає нова проблема. Коли логіка управління мережею розподіляється між декількома контролерами, ці контролери повинні бути синхронізовані, щоб забезпечити узгодженість в усій мережі. Залежно від частоти синхронізації контролерів,

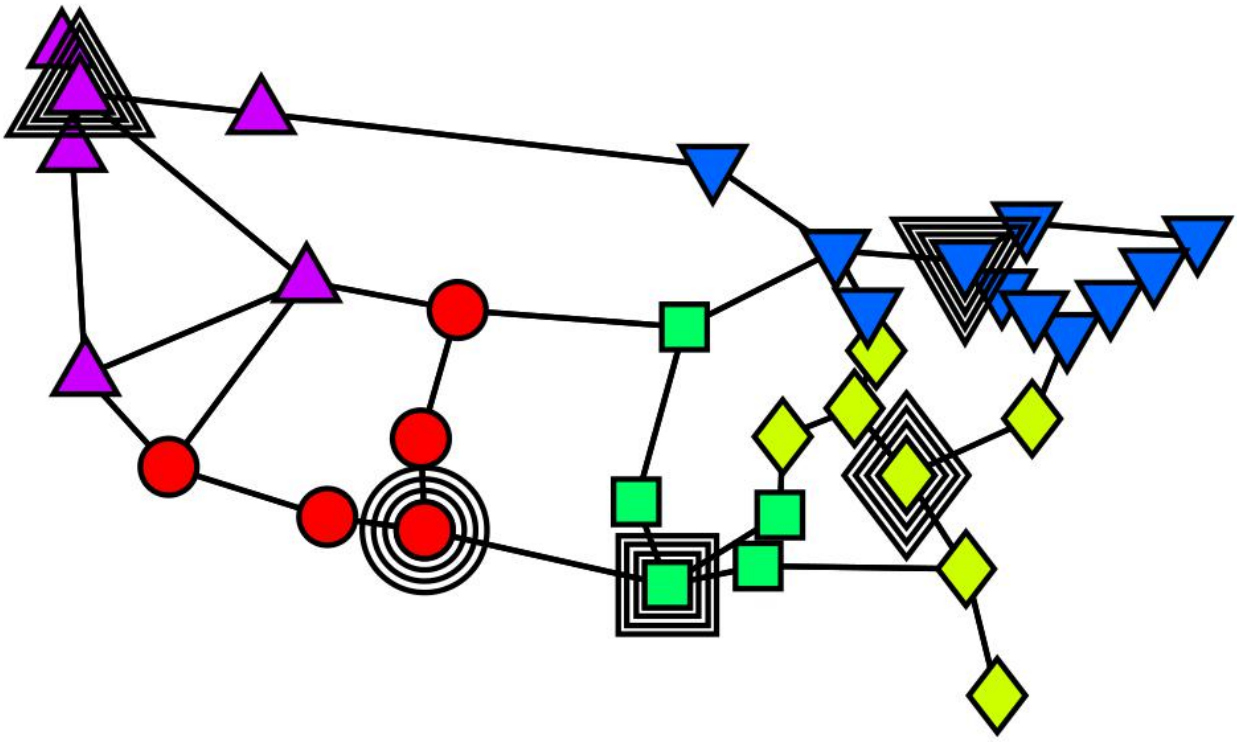


Рис. 1.8. Кластеризація мережі не урахувуючи навантаження на контролери

затримка передачі між контролерами може відігравати важливу роль. На малюнку. 1.9 показано мережеві контролери: відстані між ними не менше 50% від діаметра мережі.

1.3. Деякі способи вирішення проблеми розміщення контролерів

1.3.1. K-середніх метод

Метод k-засобів, розглянутий B. Heller et al. у 2012 році [5], він зосереджувався на швидкості захоплення мережі. Поломка мережі та перевантаження контролерів не враховуються при вирішенні проблеми розгортання контролерів.

Фактично, цей метод передбачає застосування k-середніх у кластерному рішенні із середньою затримкою (мінімально k-медіанна задача) або, із затримкою в гіршому випадку (мінімум k-центрів). Результати розв'язання задачі розміщення контролерів за цими двома вимірами показані на рис. 1.10.

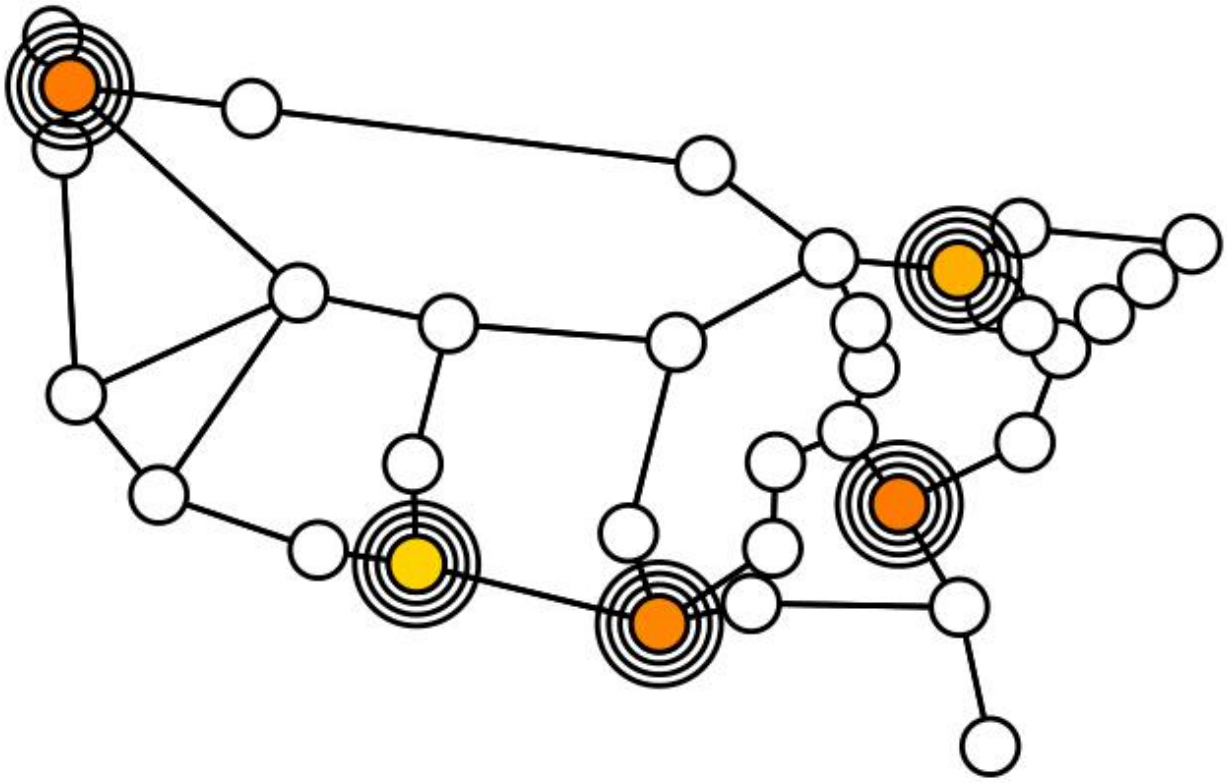


Рис. 1.9. Затримка між контролерами в мережі

Існує також варіант, коли максимальна кількість вузлів у межах заданої межі затримки (тобто затримка від них до контролера менша зазначеної межі) є критерієм оптимального розміщення контролерів. Загальна версія цієї проблеми називається проблемою максимального покриття [5].

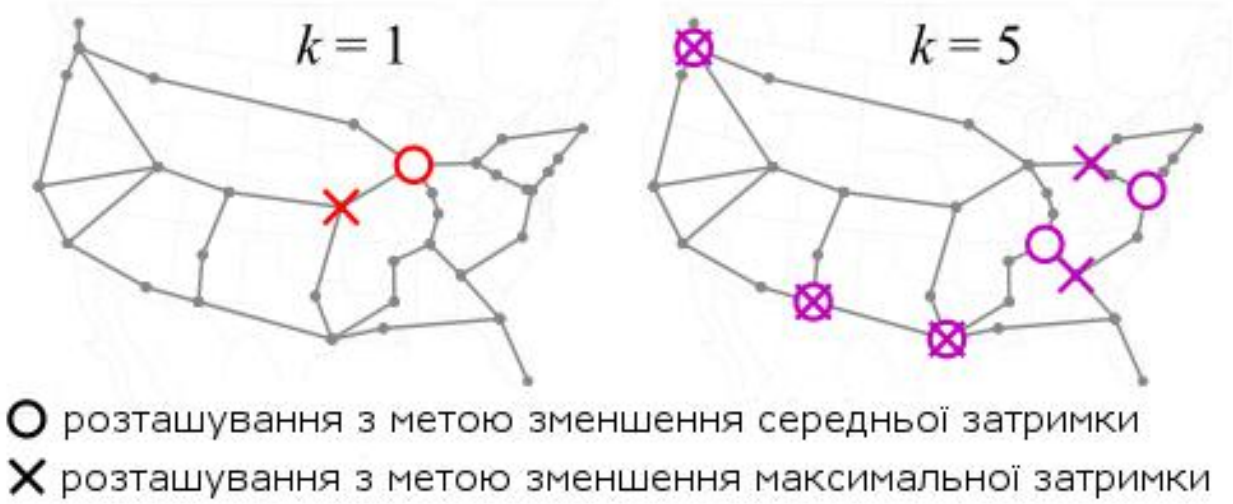


Рис. 1.10. Оптимальне розташування 1 і 5 контролерів для середньої та максимальної затримок у мережі

1.3.2. Метод розташування контролерів на основі щільності (DBCP)

У 2016 році була пропозиція використати метод розміщення контролерів на основі щільності (DBCP), який використовує алгоритм кластеризації на основі щільності для поділу мережі на кілька підмереж. Вузли тісно пов'язані всередині підмережі і менш підключені до вузлів поза підмережою, тому кожна підмережа має контролер. Розмір кожної підмережі можна визначити за потужністю контролера. Крім того, оптимальна кількість контролерів визначається в процесі кластеризації на основі щільності [6].

У DBCP мережа поділена на підмережі з згрупованими декількома вузлами. Для кожного вузла si обчислюються дві властивості: локальна щільність ρi та відстань до вузлів δi більшої щільності. Певна щільність вузлів ρi залежить від кількості вузлів у dc. Для масштабних мереж dc, як правило, визначається як 30% діаметра графу.

δi вимірюється шляхом обчислення мінімальної відстані між вузлом si та будь-яким вузлом з більш високою локальною щільністю. Для вузла з найбільшою щільністю виходить δi = maxj (dij).

Таким чином, в центрах множин є занадто високі δ вузлів.

Таким чином, після отримання центрів кластерів кожен з інших вузлів присвоюється тому ж кластеру, що і найближчий сусід який має більшу щільність.

На малюнку 1.11 показаний приклад мережевої кластеризації з використанням алгоритму, обговореного вище: мережа поділена на п'ять груп, кожна з яких має свій колір.

Після розділу ми повинні поставити контролер у кожну підмережу. Оптимальне місце розташування для кожної підмережі шукається. Розташування контролерів можна визначити відповідно до різних цільових функцій. Основними розглянутими параметрами є затримка сигналу та надійність.

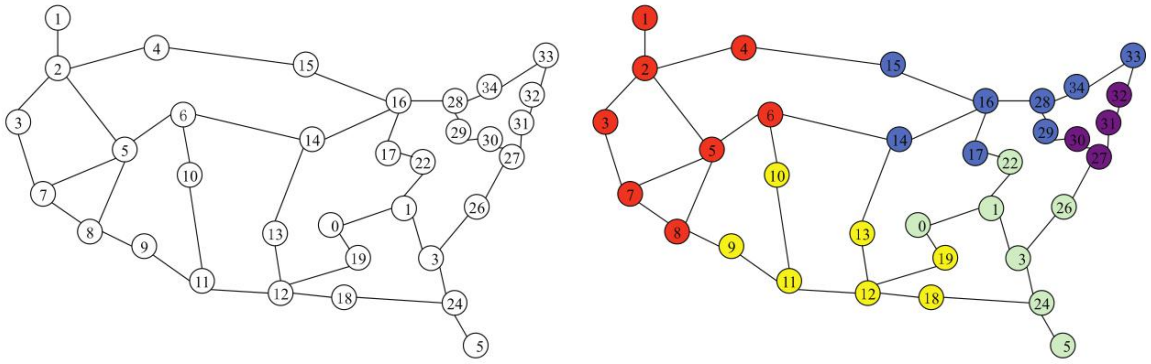


Рис. 1.11. Приклад кластеризації мережі за методом DBCP [6]

1. Затримка від контролера до робочого вузла

Для графа підмережі обчислюється середня затримка передачі сигналу збитості $\pi^{avglatency}(S(\theta))$ між контролером та вузлами S (n), керованими контролером у положенні v. Крім того обчислюється максимальна затримка підмережі $\pi^{maxlatency}(S(\theta))$.

2. Затримка між контролерами.

Щоб зменшити витрати на обмін інформацією між контролерами, вони повинні розміщуватися якомога ближче один до одного. Затримка сигналу між контролерами повинна бути мінімізована, але розташування кожного вибирається незалежно. Тому заздалегідь розрахувати відстань між контролерами неможливо.

Щоб мінімізувати затримку від одного контролера до іншого, нам потрібно розмістити його якомога ближче до інших підмереж. Можна розрахувати затримку від контролера до вузлів інших підмереж. Таким чином, мінімальна затримка від одного контролера до іншого обчислюється відповідно до його розташування відносно вузлів інших підмереж.

1.3.3. Pareto оптимальне розгортання контролерів (POCO)

Метод стійкості до відмов Парето-оптимального розміщення контролерів (POCO) пропонований в 2013 році D. Hock et al. Розгортаючи контролери SDN, метод має на меті врахувати найкраще критерії відмовостійкості мережі [7]. У багатьох топологіях, де одного контролера достатньо з точки зору пропускної здатності, вимоги стійкості вимагають їх

більшої кількості. Цей алгоритм також враховує такі показники, як затримка передачі між контролерами та залишки навантаження різних контролерів.

Основна метрика, розглянута в алгоритмі POCO, - π^{max} затримка: максимальна затримка від управління в мережі до вузла. Така затримка обчислюється як у разі коли відмови контролера в мережі не відбувається, так і у разі відмови контролерів. Якщо контролер перестає працювати, вузли, якими він керує, делегуються іншим контролерам, тим самим збільшуючи затримку для вузла контролера.

Для відмов від 0 до k-1 побудована серія C-сценаріїв для розміщення k контролерів, включаючи всі можливі комбінації. Максимальна затримка в цьому наборі визначається затримкою π_C^{max}, а за відсутності відмов - затримкою π_0^{max}.

Має сенс для контролерів бути максимально рівно розташованими, коли вони дотримуються принципу мінімізації затримки π_0 ^{max} в мережі. В тому випадку де мета - мінімізувати затримку π_C^{max}, контролери розташовані в центрі мережі.

Алгоритм POCO посилається на розташування контролерів, де ці два параметри - π_0^{max} затримка та π_C^{max} затримка - оптимальні для Парето. Враховуючи масштаб, функції та доступні технічні засоби, алгоритм повертає багато можливих рішень, залишаючих вибір найбільш підходящих для певної мережі. На малюнку 1.12 показаний приклад набору рішень, отриманих алгоритмом POCO щодо затримки π_0^{max} та затримки π_C^{max} (Парето-оптимальні рішення позначені крапками, з'єднаними чорною поліною).

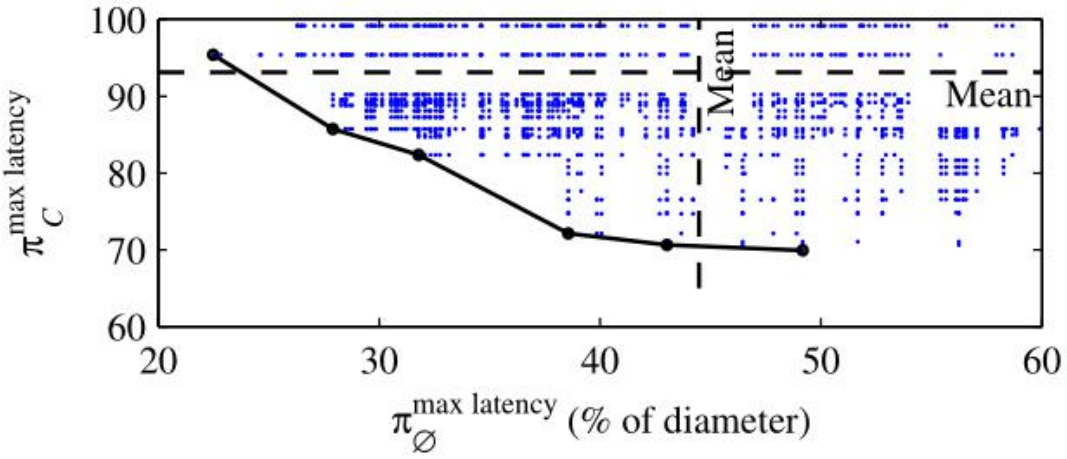


Рис. 1.12. Приклад множини рішень, отриманої алгоритмом POCO

Висновки до першого розділу

Цей розділ охоплює поняття SDN та основні особливості мереж SDN. Пояснює походження та важливість кластеризації мережі SDN.

Сформульована проблема кластеризації мережі SDN. Розглядаються завдання та проблеми, які вона повинна враховувати, та критерії якості її вирішення. Зокрема, показано, що на контролери існує важливість балансу навантаження при розробці алгоритму, який вирішує проблему кластеризації мережі SDN.

Досліджено різні підходи до кластеризації мережі SDN; для вирішення цієї проблеми були введені деякі запропоновані в останні роки методи, включаючи метод k-засобів, регулювання на основі щільності та регуляторне регулювання на основі Парето.

На основі аналізу існуючих методів кластеризації можна зробити висновок, що для вирішення проблеми розвантаження контролерів необхідно розробити модифікований метод кластеризації мережі SDN, моделювати запропонований метод, протестувати програмне забезпечення та оцінити ефективність алгоритму.

РОЗДІЛ 2.

РОЗРОБКА СПОСОБУ КЛАСТЕРИЗАЦІЇ МОБІЛЬНОЇ МЕРЕЖІ SDN

2.1. Математична модель процему кластеризації

Для обґрунтування заданого процесу ми використовуємо математичну модель, засновану на теорії графів [11].

Припустимо, вам надано неперевірений граф $G = (V,E,W,C,P)$, де V не є порожнім набором (відповідає набору вузлів мобільної мережі, де розташовані маршрутизатори), де n є порядком графа; $E = \{(i,j) \in V \times V\}$ - набір ребер (відповідає набору з'єднань між вузлами мобільної мережі); $W{:}V \to R$ - вагова функція, яка присвоює кожній вершині певну кількість ($w_i > 0$ - вага вершини $i \in V$ відповідає часу затримки маршрутизатора); $C{:}E \to R$ - функція, яка присвоює кожному ребру дійсне число ($c_{ij} > 0$ - вага ребра $(i,j) \in E$ відповідає проходу каналу); $P{:}E \to$ це функція, яка дає кожній стороні реальне число з діапазону $(0 < c_{ij} \leq 1$ймовірність того, що край буде працювати плавно).

Тоді поділ графа на під графи означає, що початковий граф G подається у вигляді набору частин (блоків) $V_1,V_2,..,V_r(r > 1)$, таких, що $V_1 \cup V_2 \cup .. \cup V_r = V; V_k \cap V_l = \emptyset; k,l = 1..r; k \neq l$.

Кінець $i \in V_k$, який з'єднаний з вершиною $j \in V_l$ ребром, називається вершиною вершини. Частина, яка з'єднує кінці різних блоків, називається краєм розділу. Набори E_{kl} називаються підрозділами. Ми називаємо граф з сукупністю під графіків V_k і набором ребер $E_k = \{(i,j) \in V \vee i \in V_k, j \in V_k\}$ підграфома G_k, тобто $G_k=(V_k,E_k)$.

Якість поділу графа можна оцінити за одним або декількома критеріями, які є параметрами параметрів підрозділів E_{kl} ($\sum_{k<1} f(E_{kl}) \to$

$min(max)$,), вершин $V_1,V_2,..,V_r$ ($\sum_{k=1}^{r} g(V_k) \to min(max)$) та підрозділів $G_1,G_2,..,G_r$ ($\sum_{k=1}^{r} h(G_k) \to min(max)$).

Отримане рішення також може застосовувати деякі обмеження (граничні умови), які необхідно враховувати при пошуку розділу. До них відносяться обмеження щодо наступних значень (верхнє та / або нижнє): кількість підграфів; кількість кінців, що входять до підграфів; обмеження балансу, тому всі підмножини $V_1,V_2,..,V_r$ повинні бути приблизно однакового розміру; максимальний дисбаланс між кінцями частин (підграфів); загальна вага кінців підграфів; ймовірність виходу з ладу всіх країв зрізів; середня кількість ребер, що входять до надрізів; середня кількість ребер, що входять до надрізів; середня вага ребер, що входять до надрізів; характеристики граничних кінців підграфів.

Під час поділу мережі на зони маршрутизації, якщо ви хочете мінімізувати загальну пропускну здатність зони з'єднання, а розмір домену повинен відповідати заданим вимогам балансу, модель буде виглядати приблизно так:

$$\sum_{kl} f(E_{kl}) = \sum_{kl} \sum_{(i,j) \in E_{kl}} c_{ij} \to min,$$

$$|V_k| \geq (1 - \varepsilon_V)\lfloor \frac{|V|}{r} \rfloor, k = 1..r.$$

Де параметр дисбалансу $0 < \varepsilon_V < 1$, який обмежує відхилення порядку алтерграфа (домену) менше середнього рівня.

Якщо мережа розділена на зони, ймовірність того, що всі перебої не відбудуться є мінімальною, а діаметр під графіків не повинен перевищувати задане значення Δ_c, тоді модель буде виглядати приблизно так:

$$\sum_{kl/|E_{kl}| \neq 0} f(E_{kl}) = \sum_{kl/|E_{kl}| \neq 0} \prod_{(i,j) \in E_{kl}} (1 - p_{ij}) \to min,$$

$$d(G_k) \leq \Delta_c, k = 1..r.$$

.

Цільова функція відповідної моделі полягає в наступному, якщо загальний час затримки на граничних кінцях зведений до мінімуму:

$$\sum_{kl/|E_{kl}| \neq 0} \sum_{i \in V_{kl}} \omega_i \rightarrow min$$

Таким чином, поділ мережі на зони маршрутизації (оптимально) дозволяє збалансувати навантаження між зонами та мінімізувати ймовірність виходу з ладу прикордонних маршрутизаторів.

2.2. Основні принципи побудови методу кластеризації

Будемо вважати, що топологія мережі $G(S,L)$ буде складатися з багатьох маршрутизаторів S і багатонаправлених зв'язків між ними L. Зауважте, що якщо між вузлами мережі є прямий зв'язок, довжина з'єднання однакова, якщо немає — рівна нулю

На відміну від інших методів, цей алгоритм аналізує топологію мережі, що ділиться в підмережі. Загальними є вузли, які мають найпоширеніші з'єднання, - це ті, що мають більше з'єднань в середині підмережі, і такі, що мають менше з'єднань в інших підмережах, тому ці підмережі повинні бути розділені як окрема мережа. Після спільного використання вищезазначеного, підмережа має менш високий показник відмов. Затримка підмережі також зменшується за допомогою тонкозернистої кластеризації.

Загалом, алгоритм складається із трьох основних етапів:

1) проаналізувати мережеву топологію, щоб розподілити щільність з'єднання між маршрутизаторами

2) відповідно до виявлених значень відстані від вузла з більшою щільністю та поточного вузла розподілити мережеві маршрутизатори;

3) вирішити задачу щодо вибору місця для контролера у кожній підмережі відповідно до визначених критеріїв.

Переваги розробленого алгоритму кластеризації охоплюють такі аспекти. По-перше, ви можете вибрати оптимальну кількість контролерів для певної мережі. По-друге, побудова мережі топологічних з'єднань на основі щільності знижує ймовірність розгортання контролерів у вузлах з високим ризиком виходу з ладу. По-третє, проблема розміщення великої кількості контролерів у мережі зводиться до проблеми розміщення в мережі контролера, що зменшує складність обчислення.

2.3. Метод кластеризації

Топологія мережі складається з багатьох маршрутизаторів.

В алгоритмі поділ мережі відбувається за рахунок поділу множини. Для кожного маршрутизатора s_i обчислюються два значення: локальна щільність ρ_i і відстань до вузла з більш високим значенням локальної щільності δ_i (пріоритет вузла для завдання кластера). Ці значення залежать лише від зв’язків між маршрутизаторами.

Локальна щільність ρ_i для кожного маршрутизатора обчислюється за формулою, яка використовує коефіцієнт зв'язування k_{cl} та відстань між маршрутизаторами d_{ij}.

$$\rho_i = \sum_j k_{cl}(d_{ij} - d_c) \quad (2.1)$$

Максимальне значення відстані роутера поблизу d_c - обмежене значення відстані, яке вузли мережі можуть "наблизити" до поточного маршрутизатора. Коефіцієнт підключення k_{cl} дорівнює 1, якщо є позитивна різниця між опорною межею відстані та відстані між маршрутизаторами.

Для розподілених мереж з більш ніж 50 вузлами ефективно використовувати максимальне значення відстані маршрутизатора поблизу як 0,3 діаметра мережі.

Відстань до вузла з найвищим значенням локальної щільності δ_i обчислюється як мінімальна відстань до вузла з найвищим значенням локальної щільності:

$$\delta_i = \min_{j:\rho_j > \rho_i} (d_{ij}) (2.2)$$

Для вертикальних точок з найбільшим значенням локальної густини максимально приймається значення відстані до вузла з найвищим значенням локальної щільності, такі кінці вибираються спочатку як центри заданих підмереж.

Детальний псевдокод алгоритму аналізу розподілу щільності зв'язків у мережевій топології показаний на рисунку 2.1. Алгоритм також вибирає рекомендовану кількість вузлів, які слід вибрати як окремі задані центри.

```
int analyzeDensity(G = (S, L), dc) {
    int k = 0;
    for (s: S) {
        ro[s] = getNumberOfNodesWithinDistance(s, dc, G);
    }
    for (s: S) {
        delta[s] = minDistanceToHigherDensityNode(s, ro, G);
        delta_Avg = sum(delta[0], ..., delta[S]) / S;
        if delta[s] > delta_Avg {
            k++;
        }
    }
    return k;
}
```

Рис. 2.1. Псевдокод аналізу розподілу щільності зв'язків по мережевій топології

Це створює чергу маршрутизаторів, з яких можна вибрати вузли. До переваг цього методу можна віднести можливість вибору будь-якої кількості маршрутизаторів підряд. Якщо кількість наборів, що підлягають вибору, менше рекомендованого значення для кількості наборів , для набору будуть обрані вузли з більшим шляхом до вузла з більш високою локальною щільністю. Якщо кількість наборів, що вибираються, перевищує рекомендоване значення заданого числа , спочатку переважні вузли з

більшим пріоритетом, а потім вузли з більш високою локальною щільністю вибираються між вершинами одного пріоритету.

Потім кожна вершина, підключається до того набору, в якому центральний вузол максимально наближений до даного. Цей алгоритм не враховує навантаження трафіку та кількість вузлів, які можуть бути предметом контролера кластера. Детальний псевдокод алгоритму підключення вузлів до кластерів показаний на малюнку 2.2.

```
void clustering(Graph G = (S, L), ro, delta, k) {
    clusterCenters = findClusterCenters(k);
    for (s: S) {
        if clusterCenters.contains(s) {
            cluster = new Cluster();
            cluster.add(s);
        }
        else {
            cluster = findNearestHigherDensityNode(s);
            cluster.add(s);
        }
    }
}
```

Рис. 2.2. Псевдокод задачі кластеризації для мережі

2.4. Вирішення проблеми розгортання контролера для підмережі

Останній крок у прийнятті рішення про розділення мережі SDN на підмережі - це вибір місця для контролера кластерів. Кожен кластер має лише один контролер. Завдяки підходу, обговореному в попередніх пунктах, проблема багатофакторного розміщення декількох контролерів може бути замінена проблемою оптимального розміщення контролерів у кожній підмережі.

Ви можете використовувати кілька заходів для пошуку контролера, включаючи багаторазові затримки трафіку, затримки трафіку між контролерами та допущення до помилки підмережі, коли одне або більше з'єднань не вдається. Розроблений алгоритм буде використовувати вимірювання затримки трафіку в межах заданої підмережі, особливо із середнім та найгіршим значенням.

Середня затримка трафіку для контролера на вузлі підмережі для графу підмережі може бути обчислена за такою формулою залежно від довжини з'єднання між контролером та поточним вузлом кластера залежно від тривалості з'єднання між ними.

$$\pi^{avglatency}(S(\theta)) = \min_{v \in S(\theta)} \frac{1}{|S(\theta)|} \sum_{s \in S(\theta)} d(v,s) \quad (2.5)$$

Цільову функцію для найгіршого значення затримки трафіку для контролера, коли він знаходиться у вузлі підмережі, можна обчислити за наступною формулою.

$$\pi^{worlatency}(S(\theta)) = \min_{v \in S(\theta)} \max_{s \in S(\theta)} d(v,s) \quad (2.6)$$

2.5. Оцінка обчислювальної складності розробленого алгоритму

Оцінюючи часову складність алгоритму, можна зрозуміти, що цей процес можна розділити на чотири частини: обчислення щільності для кожного вузла, знаходження найближчого маршрутизатора, кластеризація та знаходження розміщення контролерів.

Складність часу обчислення для кожного вузла залежить від значення d_c та загальної кількості маршрутизаторів n у мережі. Високе значення d_c призводить до високої середньої щільності $\acute{\rho}$ та великого часу пошуку, що можна оцінити $O(\acute{\rho}n)$ як часову складність фази.

Часову складність пошуку маршрутизатора з найменшою відстанню з більшою щільністю можна приблизно оцінити $O(n)$. Це пояснюється тим, що більшість маршрутизаторів близькі до маршрутизаторів вищої щільності, і

```
void clusteringWithLoad(Graph G = (S, L), ro, delta, controllerCapacity)
{
    for (s: S) {
        delta[s] = getBorderline(s, G);
    }
    S.sortBy(delta);
    for (s: S) {
        n[s] = cluster(s);
    }
    for (s: S) {
        ul = getNeighbors(s, (ro > ro[s]));
        ul.descendingSortBy(ro);
        for (s1: ul) {
            si = union(n[s], n[s1]); //si – всі ребра, що належать до n[s]
або n[s1]
            if (controllerCapacity >= sum(l(si))) {
                n[s1].add(s);
                break;
            }
        }
    }
}
```

Рис. 2.3. Псевдокод для мережі з урахуванням ліміту завантаження кластера

лише невелика їх кількість знаходиться далі від маршрутизаторів підвищеної щільності.

Розраховуючи складність часу для фази розроблення кластерів, слід зазначити, що алгоритм проходить через усі мережеві вузли і вузол призначається тому ж кластеру, якому належить сусідній маршрутизатор з найбільшою щільністю. Отже, часову складність заданої фази можна оцінити $O(n)$.

Вибір найкращого місця для контролера вимагає від алгоритму пройти всі можливі позиції, які слід враховувати в середині набору, в гіршому разі часова складність процесу $O(n^2)$.

Загальну часову складність вдосконаленого алгоритму можна оцінити $O(\acute{\rho}n + n + n + n^2)$. Був проведений експеримент для порівняння трудомісткості вдосконаленого алгоритму з іншими, результат якого наведений у розділі 4.1.

2.6. Приклад методу кластеризації

Приклад алгоритму розроблення кластерів для графу, показаного на рисунку 2.4.

Граф складається з 34 вузлів і 41 ребра між ними (кожен вузол - це маршрутизатор, і кожне з'єднання являє собою фізичну лінію зв'язку).

У цьому випадку ми припускаємо, що відстані між маршрутизаторами однакові і рівні 1, тому не станемо враховувати їх у прикладі. Як результат, найменша відстань між маршрутизаторами може бути обчислена із загальної кількості з'єднань на дорозі між маршрутизаторами. Ми приймаємо 2 одиничні відстані як максимальне значення відстані маршрутизатора d_c, що знаходиться поблизу, тобто приймаються маршрутизатори на відстані не більше 2. Для обчислення використовуємо формули 2.1 та 2.2.

Аналіз розподілу щільності між мережевими маршрутизаторами показує, що маршрутизатори 18, 31, 4 і 26 мають найбільше значення відстані для вузла з більш високим значенням локальної щільності δ та відносно велике значення густини ρ.

Середнє значення відстані до вузла з більш високим значенням локальної щільності δ становить 1,26, значення відстані до вузла з більш високим значенням локальної щільності δ та більш високим значенням локальної щільності ρ для маршрутизатора 19 вище середнього, тому маршрутизатор 19 може бути обраний.

Таким чином, цю мережу можна розділити на 5 груп, що вимагає розгортання 5 контролерів. Деякі маршрутизатори, особливо маршрутизатори 32 і 26, мають схожі значення локальної щільності ρ, але різні значення відстані для вузла з більш високим значенням локальної щільності δ. Тому

маршрутизатори локальна щільність яких є меншою відносяться до тієї ж групи, що і сусідній маршрутизатор з більш високим значенням локальної щільності ρ.

Більша відстань від вузла з більш високою локальною щільністю δ вказує на те, що ці маршрутизатори можуть бути призначені концентратором для нового кластера, оскільки вони розташовані далі від центру кластера. Після призначення вузлів 18, 31, 4, 26 і 19 центрами кластерів, інші маршрутизатори можуть бути підключені до того ж кластеру з сусіднім маршрутизатором з більш високим значенням локальної щільності ρ.

Результати алгоритму поділу на кластери показані на малюнку 2.5. Всього було розділено 5 кластерів, а маршрутизатори окремого кластера були пофарбовані в інший колір. Візуальне зображення локальної щільності та розподілу відстані вузла з більшою щільністю у вузлах зображено на малюнку 2.6.

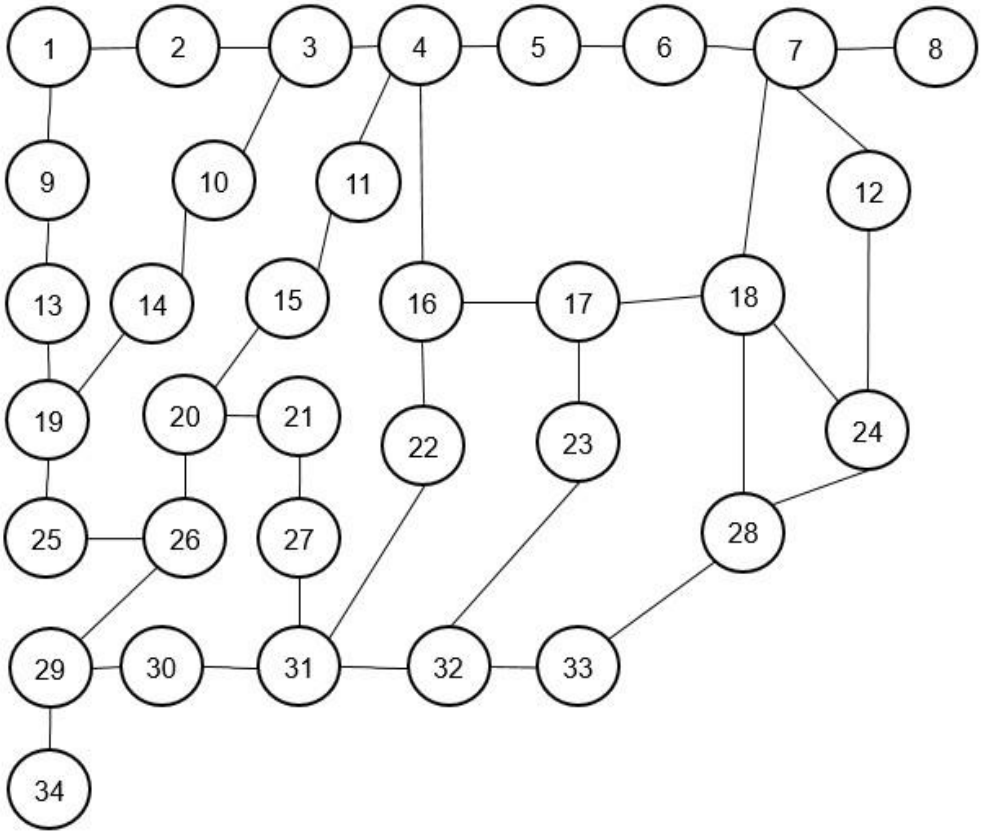


Рис. 2.4. Граф для моделювання

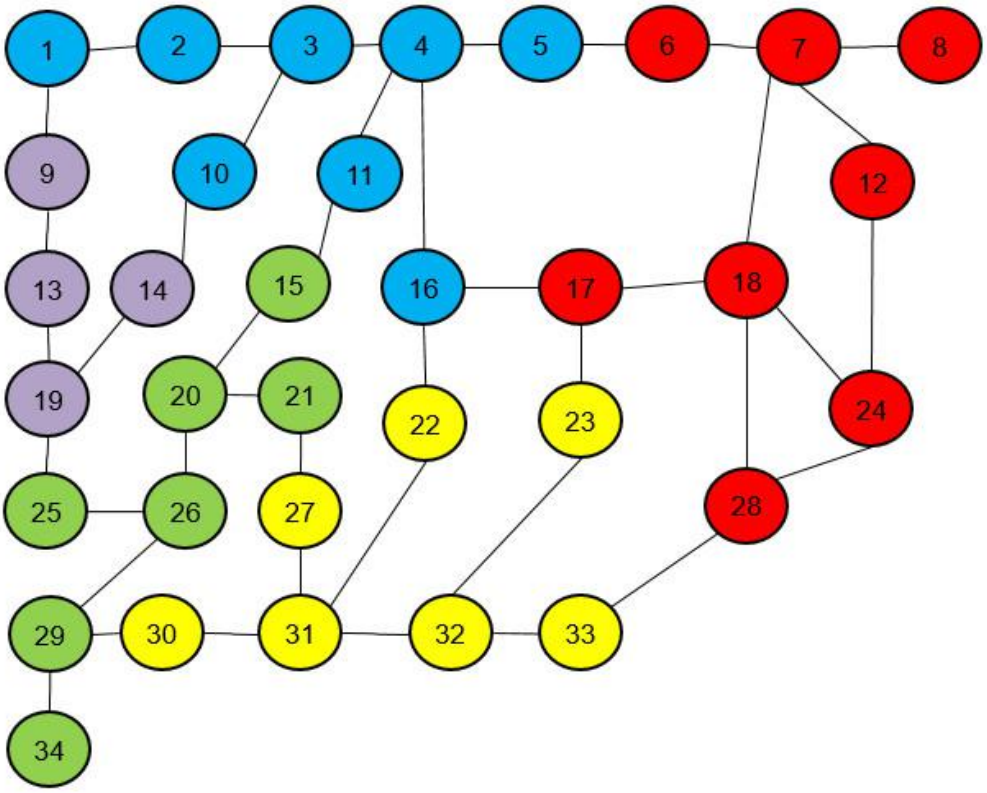


Рис. 2.5. моделювання заданого алгоритму

Рис. 2.6. Візуалізація локальної щільності та розподілу відстані по вузлах із більшою щільністю по вузлах

Таблиця 2.1

Результати заданого алгоритму

Id вузла	ρ	δ	Id вузла	ρ	δ
1	4	1	18	9	3
2	8	1	19	6	2
3	8	1	20	7	1
4	10	3	21	5	1
5	6	1	22	7	1
6	6	1	23	6	1
7	8	1	24	6	1
8	4	1	25	6	1
9	4	1	26	8	3
10	5	1	27	6	1
11	6	1	28	7	1
12	6	1	29	6	1
13	5	1	30	7	1
14	5	1	31	9	3
15	5	1	32	8	1
16	9	1	33	6	1
17	8	1	34	3	1

Висновки до другого розділу

У цьому розділі представлена математична модель задачі мережевого розбиття на кластери.

Підкреслено основні принципи побудови заданого алгоритму.

Розроблений алгоритм розбиття на кластери мобільної мережі. Розглядається проблема розміщення контролера в підмережі та вибирається рішення для його використання в розробленому алгоритмі. Таким чином, створюється алгоритм, який може вирішити задачу розміщення контролера.

Наведено приклад роботи алгоритму, що був розроблений.

РОЗДІЛ 3.

МОДЕЛЮВАННЯ РОЗРОБЛЕНОГО СПОСОБУ КЛАСТЕРИЗАЦІЇ

Для моделювання запропонованого алгоритму було обрано об’єктно-орієнтовану мову програмування Java. Цей вибір має ґрунтуватися на таких чинниках:

- Java - крос-платформова, що означає, що можна запускати заздалегідь зібрану програму, написану на Java, незалежно від платформи / операційної системи (наприклад Linux, Unix, Mac OS, Windows, Android, Web);
- великий вибір інструментів та бібліотек, які можна використовувати для програмування потрібного рішення;
- наявність зручного інтегрованого середовища розробки (особливо Intellij IDEA від JetBrains);
- обрана мова є сучасною і популярною, її використання актуально і корисно для розвитку навичок програмування.

Програма розроблювана в інтегрованому середовищі розробки, встановлюються Java JRE 8 та Java SDK 8.

Бібліотека JavaFX, яка входить до Java JDK8, використовується для візуалізації.

Працюючи з програмою, доцільно використовувати наступні технічні особливості комп’ютерної системи:

- процесор з тактовою частотою не менше 2 ГГц;
- щонайменше 1 ГБ оперативної пам’яті;
- віртуальна машина JVM встановлена щонайменше у 8 версії.

Інші функції суттєво не впливають на правильну роботу програми.

3.1. Опис логіки програми

Програмний продукт повинен мати можливість працювати з такими функціями:

- створювати, зберігати, редагувати графи мережі;
- моделювати вибраний алгоритм кластеризації;
- оглядати та аналізувати результати моделювання;
- можливість розширити можливості програми, додавши її до інших алгоритмів.
- Готуючи рішення, було зазначено, що програму слід розділити на 3 шари:
- модель, що зберігає граф системи;
- наочна демонстрація моделі;
- моделювання алгоритму;

Тому розглянемо структуру програми. Класи Model, Node і Link відповідають за рівень моделі, клас View відповідає за візуалізацію моделі, а клас Algorithm відповідає за моделювання алгоритму.

Клас вузлів відповідає за один вузол мережевого графіка. Вузол містить поля id, координати x і y центру вузла зображення вузла у перегляді та поточне значення кольору вузла та методи, які читають і змінюють значення цих полів.

Клас з'єднання відповідає за окреме з'єднання між двома вузлами графу.

Графова модель системи зберігається в класі Model. Модель, яка представляє собою сукупність прикладів класів Node та Link, має поля Node та ModelLinks відповідно. Значення наступного ідентифікатора доданого вузла базується на збільшенні поля maxIndex, забезпечуючи таким чином ідентичність ідентифікатора.

Клас моделі відповідає за додавання вузла, видалення вузла, переміщення вузла, додавання з'єднання між двома вузлами в графі, видалення з'єднання між двома вузлами в графі та зміну ваги з’єднання між

двома вузлами в мережевій графовій моделі, містить методи changeLinkWeight, addNode, removeNode, moveNode, addLink, removeLink. відповідно. Клас моделі також включає такі методи обслуговування:

- getNode повертає вузол опираючись на ідентифікатор;
- getModelLinks повертає набір посилань моделі;
- getModelNodes повертає набір модельних вузлів;
- getNodeSize повертає розмір вузла в графі колекції посилань;
- findLink, якщо таке є, повертає з'єднання між двома вузлами графіка;
- getLinks повертає набір посилань опираючись на ідентифікатор вузла, який може бути початком або кінцем цих з'єднань;
- generatorLinkBounds повертає координати початкової та кінцевої точок контактів на графовому екрані.

У класі файлів існують ствоюючий, збережуючий та відкриваючий методи, що дозволяють створити новий файл, зберегти модельний файл та прочитати модель з файлу відповідно. Клас моделі є спадкоємцем файлового класу.

Змінні startNodeId і endNodeId використовуються для зберігання ідентифікацій вузлів, оскільки вам потрібно по черзі натискати на початкові та кінцеві вузли, щоб додати та видалити з'єднання між двома вузлами в графі.

Для виконання операції натисніть відповідну кнопку ліворуч в вікні та на область налаштування графу один чи два рази (залежно від операції: додавання та видалення графового вузла вимагає натискання на область один раз, переміщення вузла та робота із посиланнями - двічі).

Після натискання на поле редагування, остаточні координати, які використовуються для надсилання даних до класу TaskModel, фіксуються в масиві clickData. Є процесорний ClickHandler, який записує обведення в області виправлення, залежно від клавіші на клавішах, має кнопку запуску,

яка перемикає програму на певний режим введення та скидає послідовний ключ ClickHandler:

0 – додати вузол;

1 – видалити папку з вузлом;

2 – змінити вагу вузла;

3 – перемістити вузол;

4 – додати посилання;

5 – видалити з'єднання;

6 – змінити вагу переходу.

Операції, які змінюють вагу (додавання / видалення) вузла / посилання при введенні ваги, використовують значення, отримане в текстовому полі textArea.

Після успішної операції входу граф видаляється з моделі за допомогою методу перефарбовування (вузли repaintNodes та з'єднання repaintLinks), читаються колекції ModelNode та ModelLinks класу Model, а також встановлюються вузли та графові з'єднання.

Програма розділена на візуальну частину та фактичну модель графу. Вся логіка, пов'язана з графом, відбувається в моделі, граф відображається за допомогою методу перемалювання класу View.

Клас GraphGenerator - це клас, який складає граф.

Параметри, що відповідають за кількість вузлів, мінімальну та максимальну вагу торців та з'єднання системи, показані в конструкторі класу. Параметр MaxClusterNode, відповідальний за найвищу кількість точок в наборі, встановлюється окремо.

По-перше, кількість груп графа формується визначеними кінцевими точками, які приблизно вдвічі перевищують кількість вузлів, і початковий поділ вузлів здійснюється за допомогою рівномірного розподілу. "Змішування" наступних наборів - перенесення певного кінця з однієї групи в іншу (метод mixClusterPointers). Отримані вказівники зберігаються в серії записів на етапі кластерних покажчиків.

Метод CheckParametres генерує ваги, знаходить загальну вагу торців та з'єднань, які потрібно будувати - Поля “Загальна вага” та “загальне посилання”.

Далі метод makeClusterLinks визначає з'єднання між кінцями, що належать різним групам, з урахуванням максимальної межі з'єднання. Для кожного вузла виконується випадкове моделювання з'єднань, кількість яких не перевищує 0,75 максимально можливих з'єднань вузла.

Метод MakeInterClusterLinks визначає з'єднання між кінцями, що належать різним групам, з урахуванням максимальної межі з'єднання.

Метод GenerateTrafficQuantity генерує випадкові, рівномірно розподілені значення для трафіку для з'єднань, за умови, що мінімальні та максимальні значення minTraffic та maxTraffic обмежені відповідно.

Клас DensityAlgorithm відповідає за моделювання самого алгоритму. Параметри топології графу вводяться, параметри навантаження контролера (з урахуванням цього обмеження в алгоритмі) maxControllerLoad та значення кількості кластерних груп, на які повинна бути поділена мережа.

Оскільки алгоритм складається з трьох етапів, моделювання алгоритму можна розділити на GraphDensity, clusterGraph, clusterLoadBalanced, setControllerPlacement - методи пошуку аномальної частини у розділенні зв’язків.

Обчислення методу аналізу GraphDensity має такі допоміжні методи, як getCloseNeighbors, getDensity, getMinDistanceIndex, getMarginityIndex, setOptimalClustersQuantity, які використовуються для пошуку сусідніх маршрутизаторів на граничній відстані до сусідніх маршрутизаторів для обчислення і визначити оптимальну кількість наборів у мережі.

На додаток до вимірювання щільності та відстані маршрутизатора, який має більш високе значення щільності minDistanceIndex, метод clusterLoadBalanced, на відміну від методу clusterGraph, використовує метрику наближеності до маргінальності кластера Index. Результати зберігаються в масиві колекції clusterNodesArray.

Залежно від обраної метрики моделювання (особливо середнього або максимального значення затримки), метод в наборі робить оптимальним положення контролера. Результати зберігаються в декількох позиціях controllerPlacementArray.

3.2. Опис інтерфейсу програми

Програма призначена для моделювання заданого алгоритму, розробленого у другій частині. Програмне забезпечення дозволяє редагувати граф та власне моделювати алгоритм.

Наступна функціональність була врахована при побудові інтерфейсу:

- можливість роботи з вузлами (додавання, виймання та перенесення ваги вузла);
- вміння працювати з посиланнями (додавати вагу, видаляти та змінювати);
- робота з файлами (завантажувати та збережігати файл).

Щоб краще зрозуміти роботу програми, необхідно описати особливості програмного інтерфейсу. Інтерфейс програмного забезпечення показаний на малюнку 3.1.

Меню розробленої програми розташоване у верхній частині головного вікна програми і складається з трьох елементів: "Файл", "Редагувати" та "Модель".

Меню файлів використовується для загального управління програмою і містить такі елементи: "Створити нове", "Зберегти", "Відкрити", "Створити граф" та "Вийти", для зручності доступні такі клавіші: Ctrl + N, Ctrl + S, Ctrl + O, Ctrl + G, Ctrl + E відповідно. Для швидкого зберігання в пам’яті користувача гарячі клавіші використовують загальну клавішу цього меню: Ctrl. Елементи Створити, Зберегти, Відкрити та Вийти використовуються для створення нового графу, збереження поточного стану графу у файл, зчитування графа із збереженого документа та виходу з програми. Елемент «Створити граф» використовується для створення графа, який відповідає

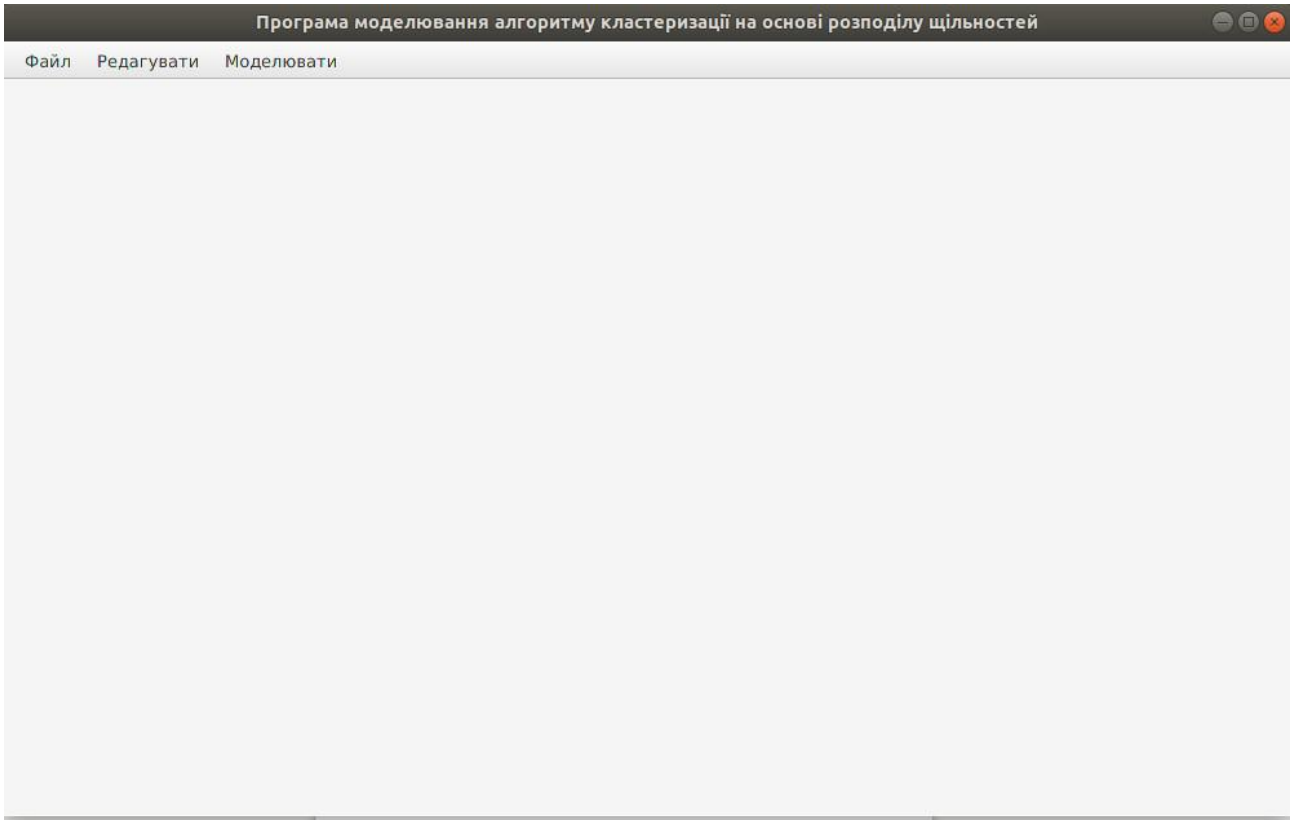


Рис. 3.1. Інтерфейс програмного забезпечення

вибраним параметрам у новому вікні. Вид меню «Файл» показаний на малюнку 3.2.

У лівій частині вікна програми є такі кнопки: «Додати вузол», «Видалити вузол», «перемістити вузол», «Додати посилання», «видалити» та «змінити вагу посилання».

Меню «Правка» використовується при створенні та редагуванні мережевого графу і включає такі елементи: «Додати вузол», «Видалити вузол», «Перемістити вузол», «Додати», «Видалити» та «Змінити вагу посилання». мова ". Ці елементи для зручності мають наступні гарячі клавіші: "1", "2", "3", "4", "5" і "6". Для зручності меню «Редагувати» має кнопки в лівій частині програми в графовій області. Зовнішній вигляд меню «Правка» показано на малюнку 3.3.

Редагування графу мережі відбувається наступним чином.

Щоб розмістити новий вузол, натисніть на пункт "Додати вузол" у меню "Редагувати" або на відповідну кнопку з лівого боку вікна програми, а

потім торкніться області мережевого графу. Середньоцентрований вузол з’явиться в регіоні. Ідентифікатор вузла автоматично генерується з кроком.

Щоб видалити, натисніть на пункт «Видалити вузол» у меню «Редагувати» або на кнопці зліва вікна програми та торкніться папки в області графу мережі. Коли ви видаляєте вузол, усі з'єднання з цим вузлом автоматично зникають.

Щоб перемістити вузол, натисніть на пункт «Перемістити вузол» у меню «Редагувати» або на кнопці зліва у вікні програми, потім торкніться графового вузла та перенесіть його на нове місце.

Щоб додати нове з'єднання між двома вузлами в графі, натисніть на пункт «Додати» в меню «Редагувати» та, також, торкніться двох вузлів у графі або відповідної кнопки. Якщо наразі між вузлами немає зв’язку, він буде доданий до моделі та відображений у графовій області.

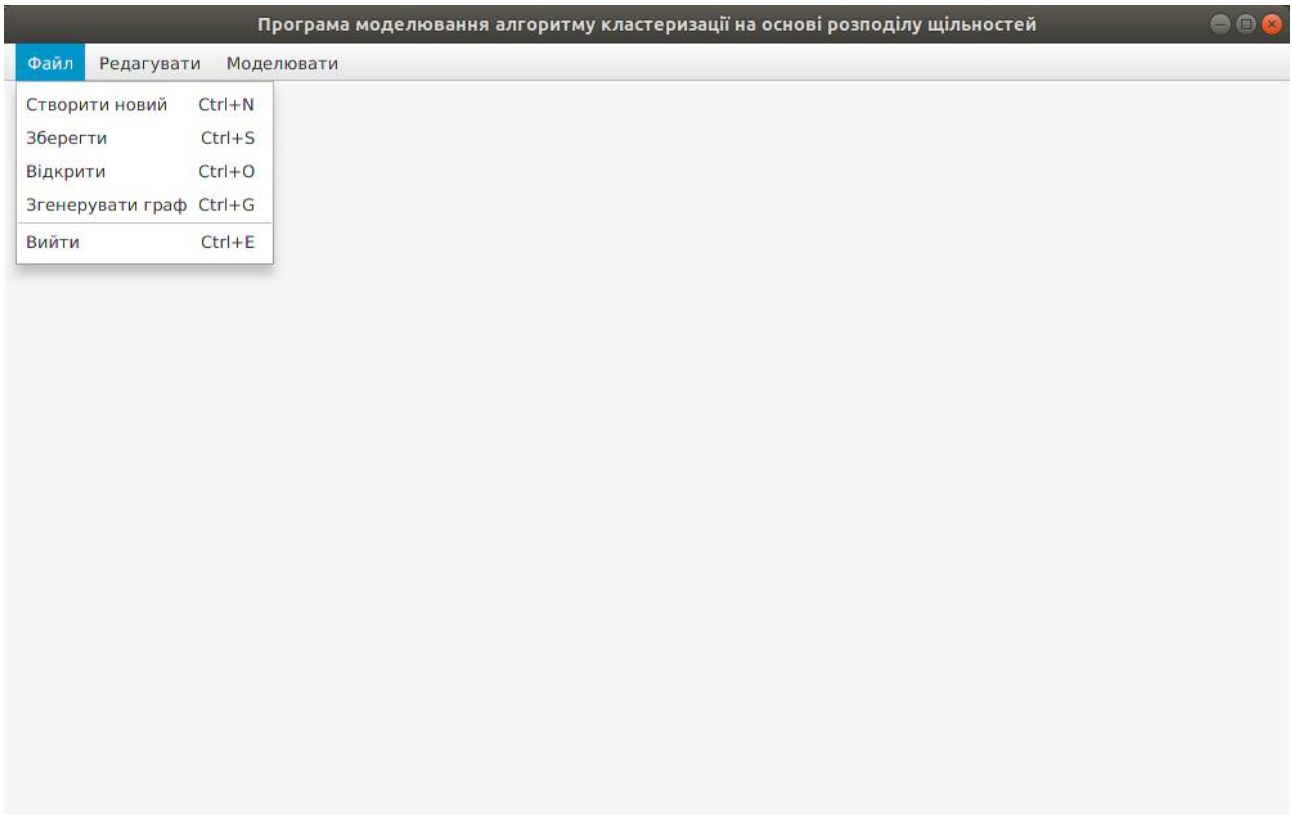


Рис. 3.2. Меню файлів

Щоб видалити нове з'єднання між двома вузлами в графі, натисніть на пункт «Видалити» у меню «Редагувати» або на відповідну кнопку зліва у вікні програми, а потім по черзі торкніться двох вузлів графу. Якщо в цей

момент між вузлами був зв’язок, він буде видалений з моделі і зникне з графової області.

Щоб відрегулювати потік трафіку між двома вузлами графа, натисніть на пункт меню «Змінити вагу контакту» або на відповідну кнопку з лівого боку вікна програми, торкніться двох вузлів графа послідовно та запишіть нову вагу зв'язку, після чого на графі буде показано нове значення для трафіку між вузлами.

Елемент, вибраний у меню "Редагувати" або натиснувши на відповідну кнопку з лівого боку вікна програми, активний, поки ви не виберете новий елемент або переключитесь на інше меню. Редагуючи граф таким чином, ви можете додати більше ніж один вузол, посилання чи перемістити більше ніж один вузол, не змінюючи режим.

Меню "Моделювання" використовується для управління імітацією і містить такі елементи: "Перевірити граф", "Перейти на необмежене моделювання", "Переключитися на обмежене моделювання", "Визначити алгоритм моделювання", "Показати області кластерів", "Показати параметри кластера", "Розміщення контролерів у наборах", для зручності є такі гарячі клавіші: Shift + C, Shift + L, Shift + E, Shift + A, Shift + R, Shift + P, Shift + K. Використовується загальна клавіша для цього меню: Shift. Зовнішній вигляд меню "Модель" показано на малюнку 3.4.

Елемент "Перевірити граф" використовується для переходу в режим моделювання. Програма перевіряє наявність вершин без зв’язків та інших помилок настройки графу.

Елементи "Перехід на моделювання без обмежень", "Перехід на моделювання з обмеженням" відповідають за вибір режиму моделювання, тобто можливість врахувати розподіл навантаження в розробленому алгоритмі. Після натискання на пункт "Визначити алгоритм моделювання" у спадному меню ви можете вибрати алгоритм моделювання, особливо той, у якому порівнюється алгоритм.

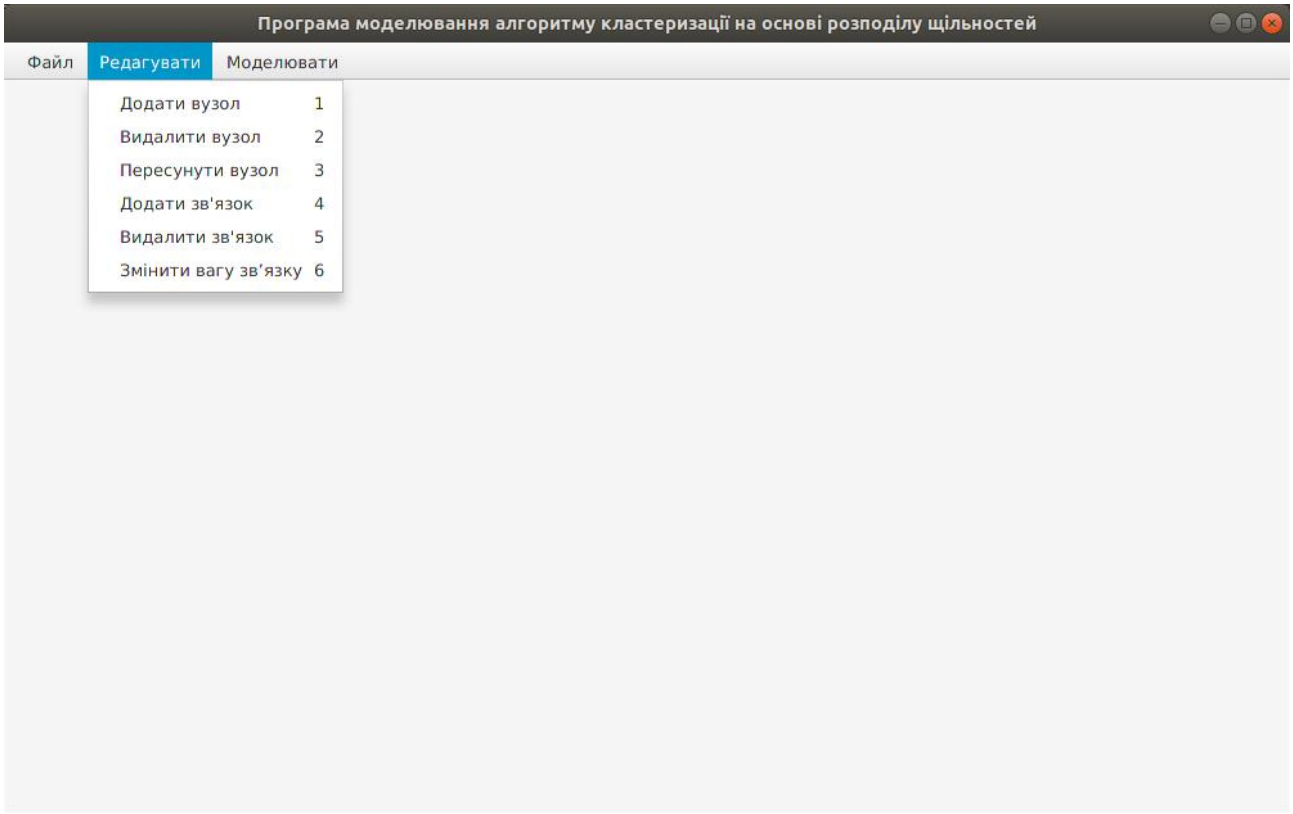


Рис. 3.3. Меню “Створити”

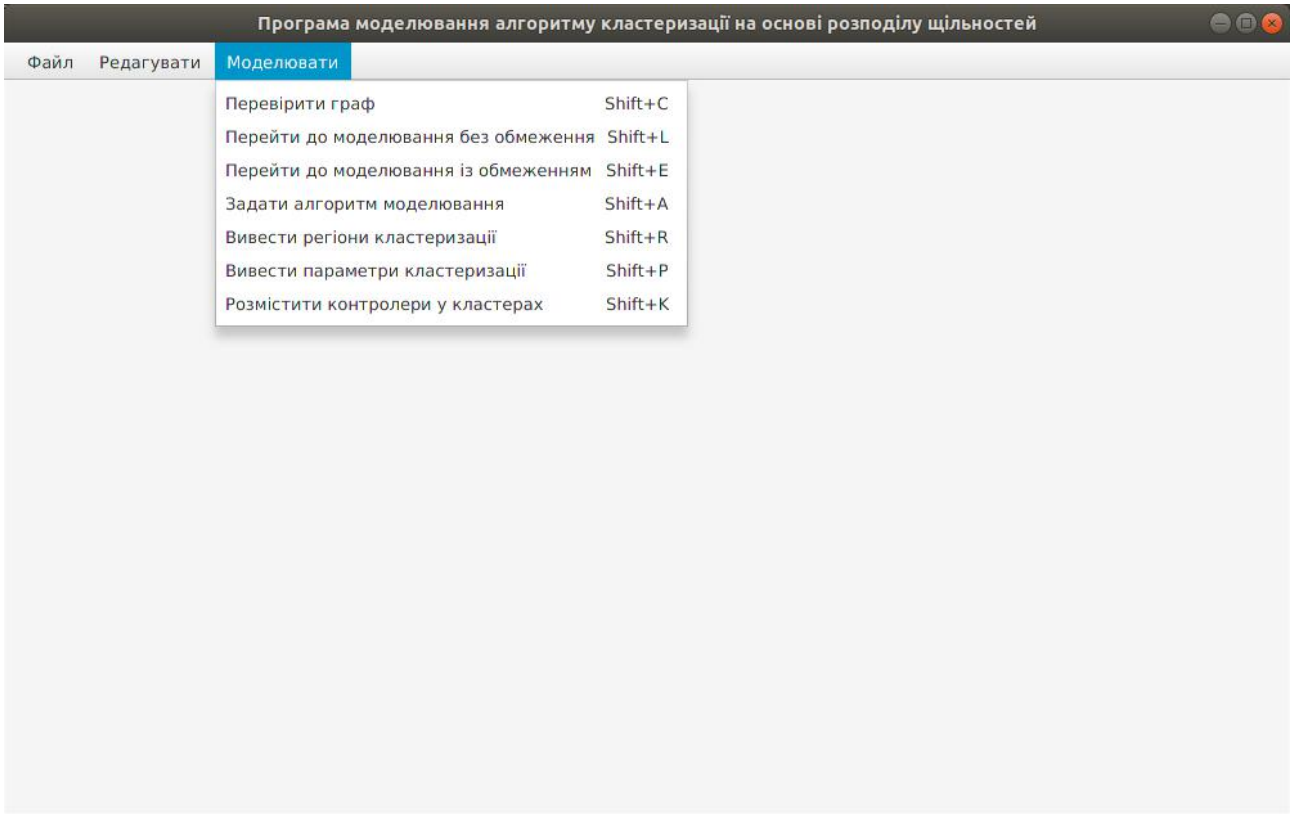


Рис. 3.4. Меню моделі

Елементи "Області кластеризації", "Налаштування кластеризації", "розташування контролерів в кластерах" відповідають за відображення результатів моделювання на екрані.

Ось зображення основних етапів програми. На рисунку 3.5 показані результати графу, на рисунку 3.6 - результати кластеризації, на рисунку 3.7 - результати кластери вершин у тексті, на рисунку 3.8 - результати обчислення параметрів моделювання, на рисунку 3.9 - результати розміщення контролерів у імітованій мережі.

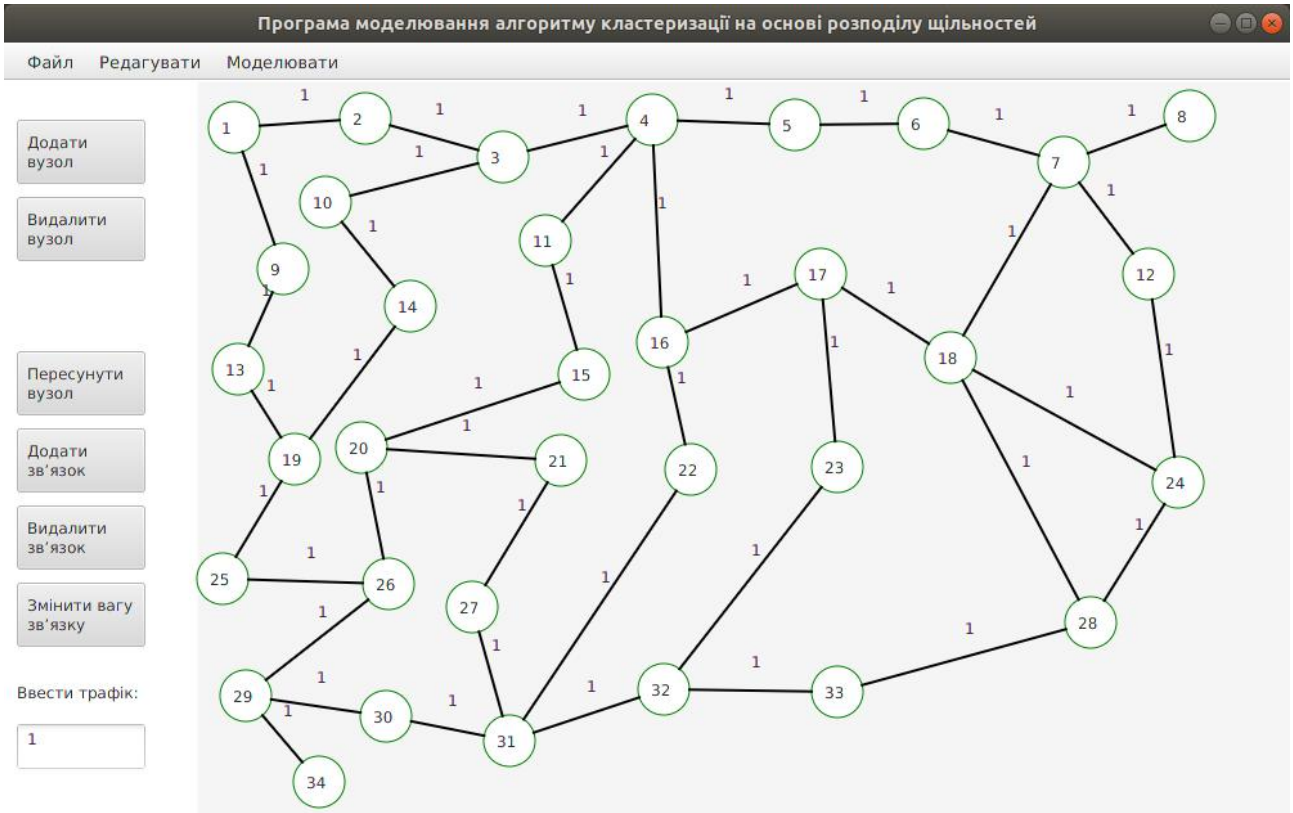


Рис. 3.5. Результат складання графу

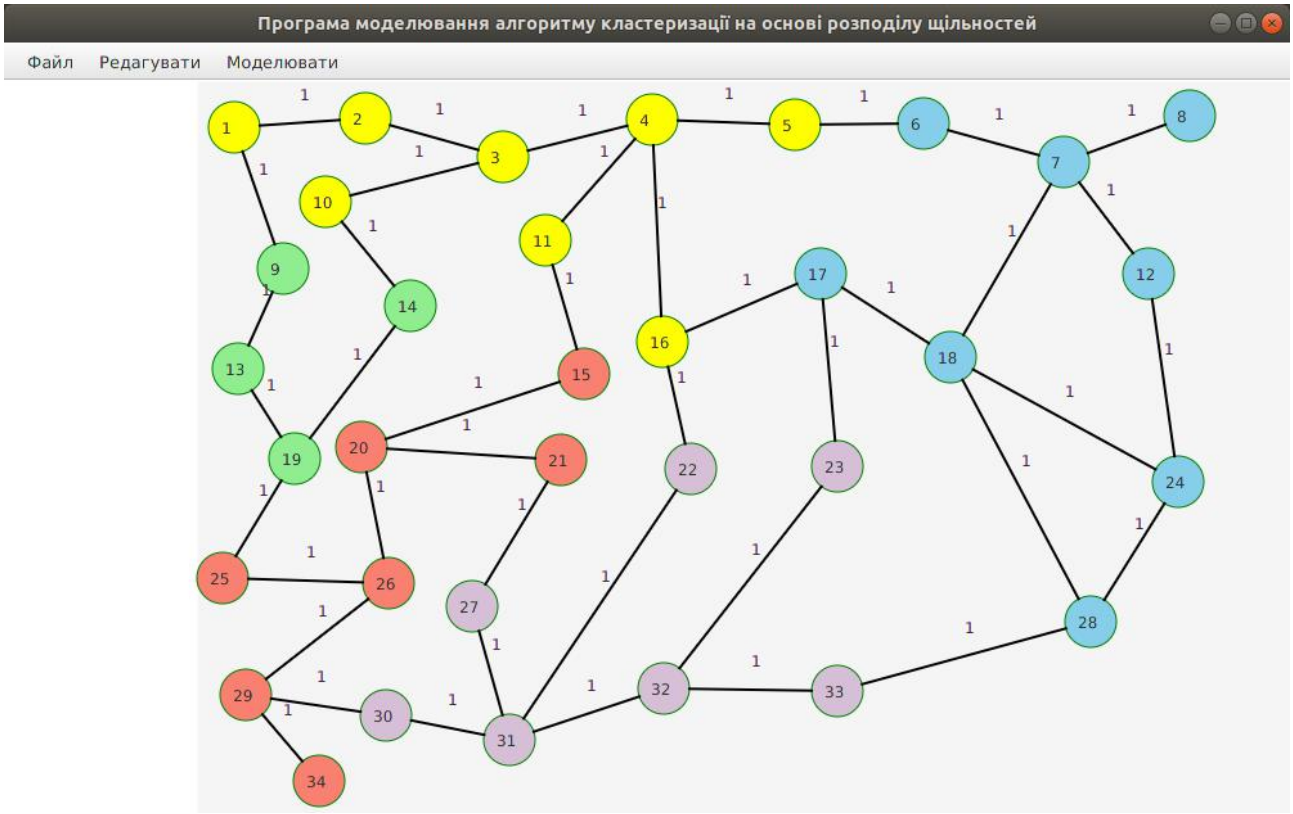


Рис. 3.6. Результат розбиття

Результати виконання кластеризації:

Кластер S1 складається з вершин {V1, V2, V3, V4, V5, V5, V10, V11, V16}.

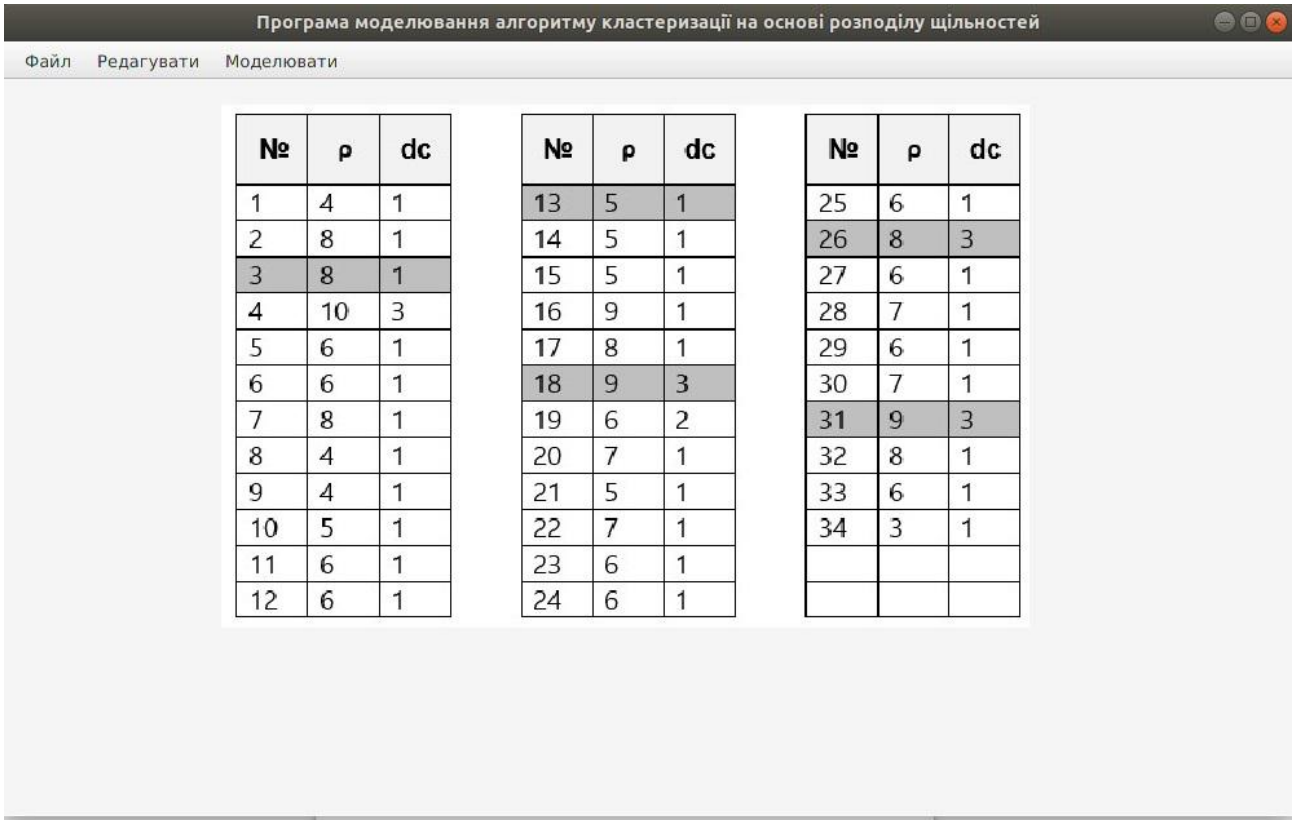
Кластер S2 складається з вершин {V6, V7, V8, V12, V17, V18, V24, V28}.

Кластер S3 складається з вершин {V9, V13, V14, V19}.

Кластер S4 складається з вершин {V15, V20, V21, V25, V26, V29, V34}.

Кластер S5 складається з вершин {V22, V23, V27, V30, V31, V32, V33}.

Рис. 3.7. Результат розбиття кінців на кластери у вигляді тексту



№	ρ	dc
1	4	1
2	8	1
3	8	1
4	10	3
5	6	1
6	6	1
7	8	1
8	4	1
9	4	1
10	5	1
11	6	1
12	6	1

№	ρ	dc
13	5	1
14	5	1
15	5	1
16	9	1
17	8	1
18	9	3
19	6	2
20	7	1
21	5	1
22	7	1
23	6	1
24	6	1

№	ρ	dc
25	6	1
26	8	3
27	6	1
28	7	1
29	6	1
30	7	1
31	9	3
32	8	1
33	6	1
34	3	1

Рис. 3.8.Результат розрахунку параметрів моделювання

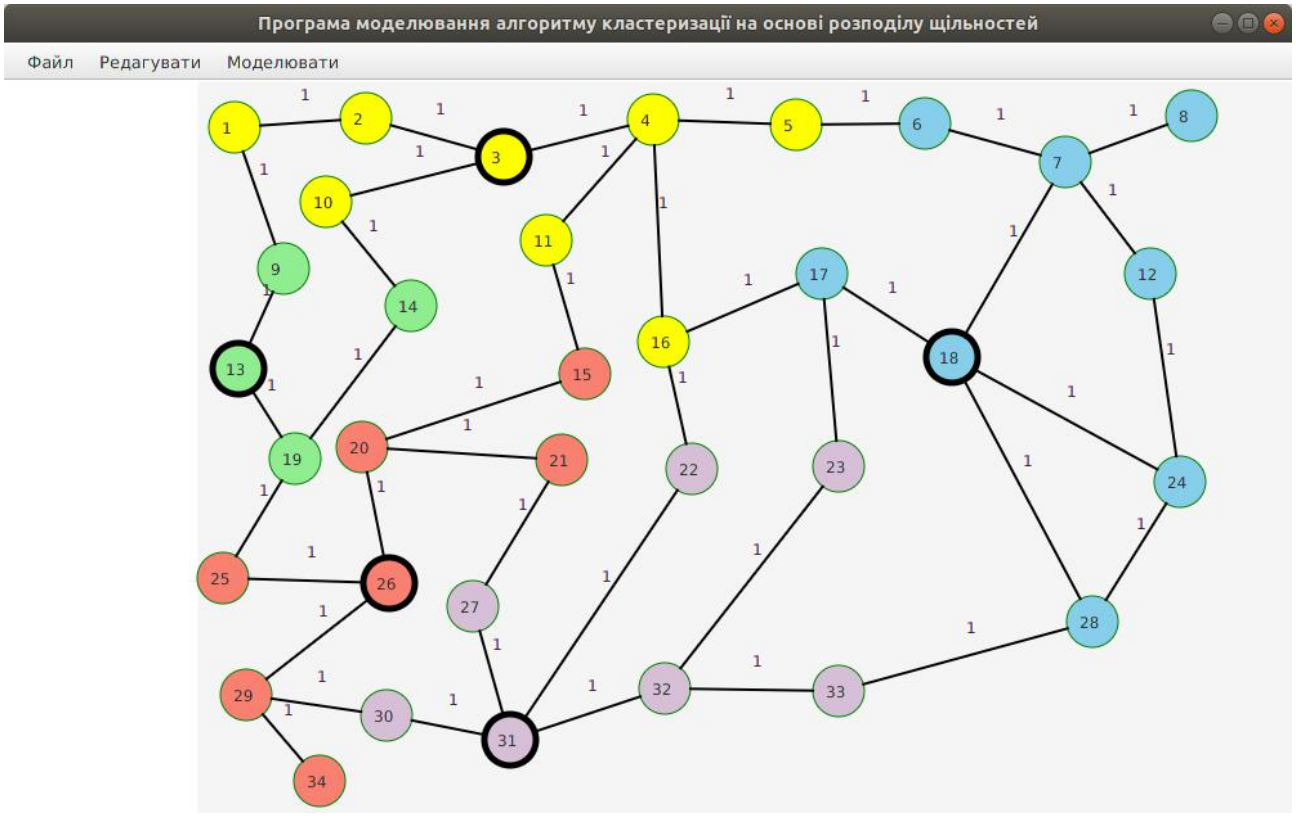


Рис. 3.9. Результат розміщення контролерів у мережі

Висновки до третього розділу

У цьому розділі розглядається та вибирається програмне забезпечення, яке використовується для програмного забезпечення алгоритму, розробленого в попередньому розділі.

Описано процес розробки моделей. Присутній опис основних класів та методів, на яких будується розроблена програма, пояснюється спосіб взаємодії, демонструється робота програми на заданому графі.

РОЗДІЛ 4.
ТЕСТУВАННЯ МЕТОДУ КЛАСТЕРИЗАЦІЇ

4.1. Аналіз результатів роботи та порівняння з існуючими алгоритмами.

Під час тестування отриманого алгоритму було виміряно кілька вимірів, які є важливими для аналізу роботи моделі. Алгоритм порівнювали з алгоритмом K-means та алгоритмом POCO.

Для тесту був використаний генератор випадкових мережевих графів, розроблений у розділі 3.

На рисунку 4.1 та таблиці 4.1 представлені результати аналізу швидкості процесу вирішення проблеми розгортання контролерів у мережі. Для аналізу використовувався графічний генератор з інкрементом 25 вузлів, всі трафікові з'єднання приймалися за 1.

Отримані вимірювання дали наступні результати. Різниця в швидкості між відносно невеликими (до 100) мережами у алгоритмів не велика, але різниця починає зростати між розробленим та алгоритмом K-means та алгоритмм POCO з поліноміальною складністю. Варто відмітити, що порядок поліноміальної складності різниці з алгоритом K-means є вищим.

На рисунку 4.2 та таблиці 4.2 представлені результати аналізу затримки розподілу сервісного трафіку залежно від кількості вузлів. Для аналізу використовувався графічний генератор з інкрементом 25 вузлів, всі трафікові з'єднання приймалися за 1.

Отримані вимірювання дали наступні результати. Запропоновані алгоритми мають різні значення затримки для різних кількісних значень вузлів, але алгоритм, розроблений у загальному індексі, має найкращі показники.

На рисунку 4.3 та таблиці 4.3 представлені результати аналізу контролю насичення контролерами топології, коли загальний трафік ведених

маршрутизаторів обмежений. Для аналізу використовували великий граф (250 вершин) і перед оглядом довелося розгорнути певну кількість контролерів.

Таблиця 4.1

Аналіз швидкості заданих алгоритмів залежно від кількості вузлів

	Алгоритм моделювання		
Кількість вершин	Розроблений	K-середніх	POCO
25	0,145	0,121	0,213
50	0,345	0,249	0,346
75	0,687	0,784	1,234
100	1,237	1,456	1,983
125	1,751	1,876	4,305
150	2,221	2,687	6,217
175	2,987	3,456	8,329
200	3,415	4,324	9,299
225	3,769	5,607	10,845
250	4,681	6,809	13,657

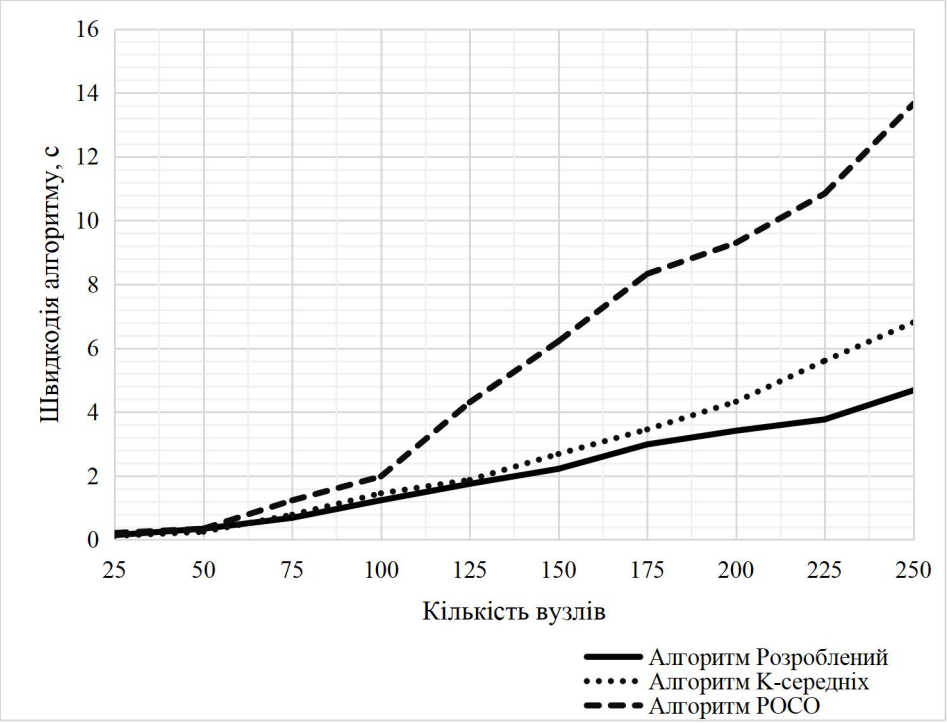


Рис. 4.1. Порівняльний графік аналізу продуктивності алгоритму кластеризації залежно від кількості вузлів

Отримані вимірювання дали наступні результати. Розроблений алгоритм дозволяє охопити створену топологію більш високим показником покриття, що свідчить про те, що алгоритм працює більш ефективно при вирішенні проблеми розміщення контролерів з обмеженим трафіком підлеглих маршрутизаторів.

На рисунках 4.4 та таблиці 4.4 наведені результати аналізу насичення топологічними контролерами, у випадку коли обмежили загальну кількість маршрутизаторів. Для аналізу використовували великий граф (250 вершин) і перед оглядом довелося розгорнути певну кількість контролерів.

Отримані вимірювання дали наступні результати. Розроблений алгоритм дозволяє охопити створену топологію більш високим показником покриття, що показує, що алгоритм працює більш ефективно при вирішенні проблеми розгортання контролерів з обмеженою кількістю підпорядкованих маршрутизаторів.

Таблиця 4.2

Аналіз затримки розподілу сервісного трафіку залежно від кількості вузлів

	Алгоритм моделювання		
Кількість вершин	Розроблений	K-середніх	POCO
25	6	7	8
50	8	7	7
75	5	6	9
100	8	12	14
125	9	8	9
150	10	9	11
175	11	9	10
200	10	11	12
225	15	18	17
250	21	20	23

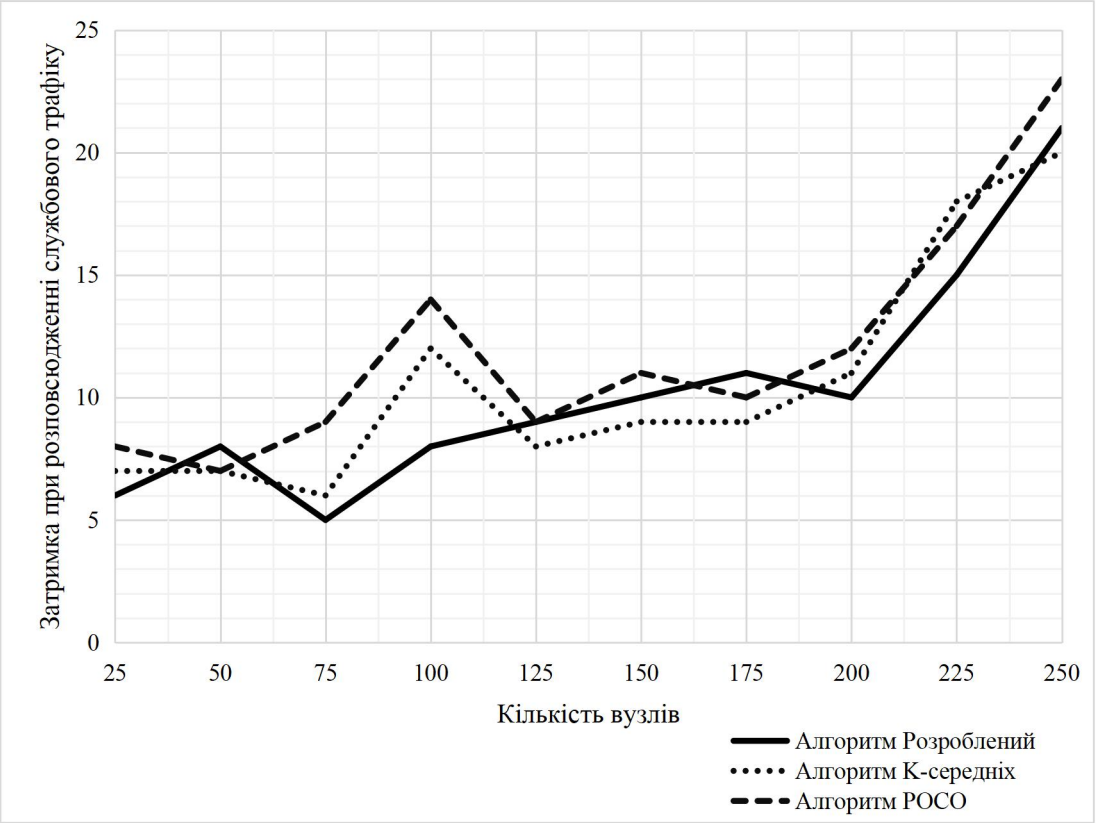


Рис. 3.2. Порівняльний графік аналізу затримки розподілу сервісного трафіку залежно від кількості вузлів

Таблиця 4.3

Аналіз насичення топології контролерами, в випадки обмеження загальний трафік керованих маршрутизаторів

	Алгоритм моделювання		
Кількість контролерів	Розроблений	K-середніх	POCO
5	0,19	0,16	0,17
10	0,24	0,19	0,22
15	0,34	0,27	0,31
20	0,43	0,37	0,39
25	0,54	0,49	0,48
30	0,69	0,595	0,58
35	0,75	0,7	0,67
40	0,91	0,84	0,82
45	0,95	0,92	0,89
50	0,98	0,94	0,95

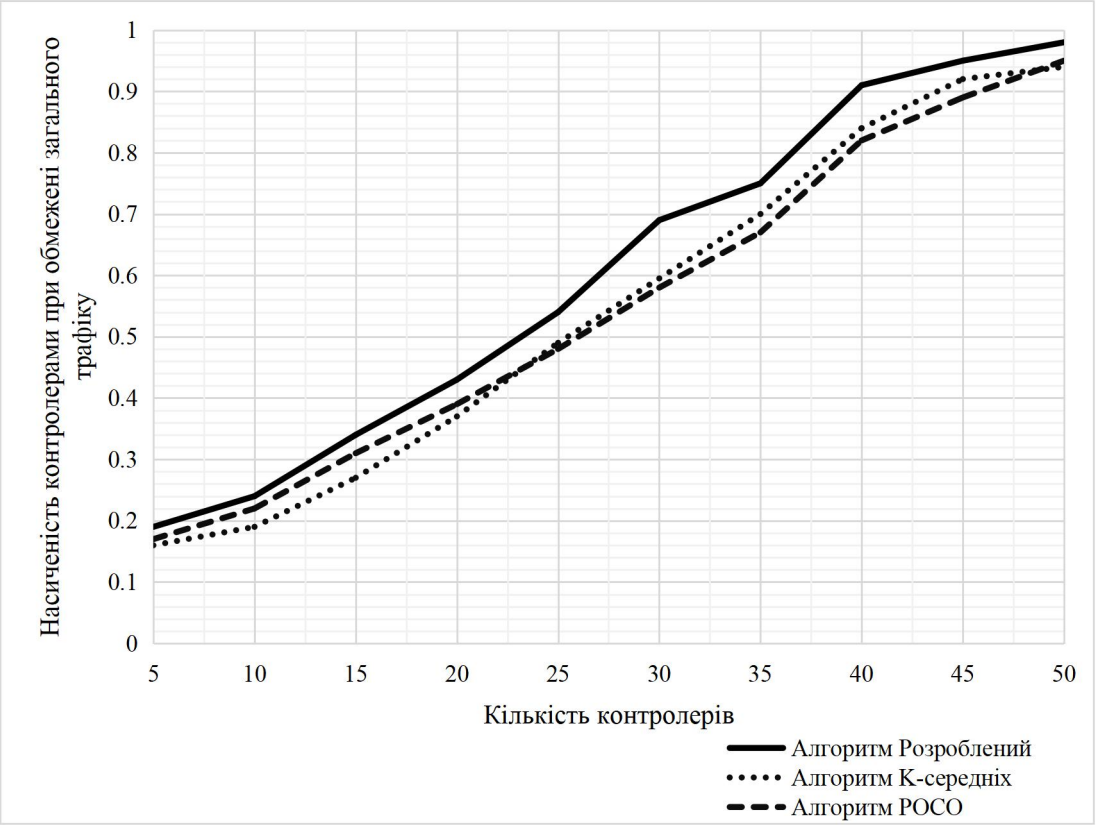


Рис. 4.3 Порівняльний графік аналізу насичення топологічним контролером, в випадку обмеження загального трафіку ведених маршрутизаторів

Таблиця 4.4

Аналіз насичення топології контролерами з обмеженням на загальну кількість табличних маршрутизаторів

	Алгоритм моделювання		
Кількість вершин	Розроблений	K-середніх	POCO
5	0,19	0,15	0,16
10	0,34	0,29	0,32
15	0,55	0,48	0,51
20	0,76	0,63	0,64
25	0,91	0,77	0,83
30	0,95	0,89	0,93
35	0,96	0,95	0,94
40	0,98	0,97	0,97
45	0,99	0,99	0,99
50	1	1	1

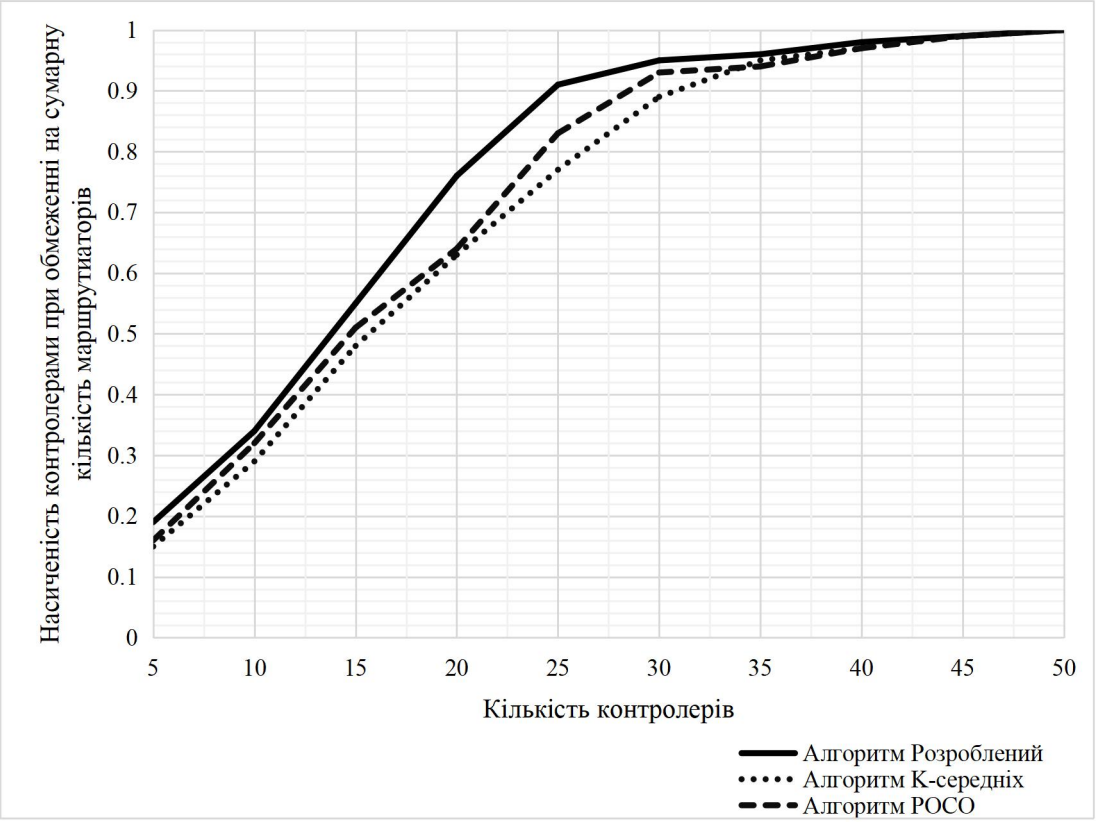


Рис. 4.4. Порівняльний графік аналізу насичення топологічним контролером з обмеженням на загальну кількість табличних маршрутизаторів

4.2. Аналіз впливу відстані близького маршрутизатора на роботу алгоритму

На малюнку 4.5 розглядається вплив одного параметра вищеописаного алгоритму на максимальне значення близької відстані маршрутизатора. Для оцінки ефекту параметра було проведено серію експериментів з графами різного розміру (від 50 до 250) з різними вимогами до кількості наборів, які необхідно виділити. та в таблиці 4.3 представлені результати аналізу насичення

топології контролерами з обмеженням загального трафіку маршрутизаторів. Для аналізу використовували великий граф (250 вершин) і перед оглядом довелося розмістити певну кількість контролерів, залежно від рекомендованого числа.

Результати показують, що значення більше при низьких і високих значеннях. Зону стійкості можна розділити між 0,3 і 0,5 діаметра d_c. Чим

більше максимальне значення відстані маршрутизатора поблизу, тим більша область пошуку, коли шукаються розташовані неподалік маршрутизатори, і значення 0,3 є оптимальним для значення діаметра.

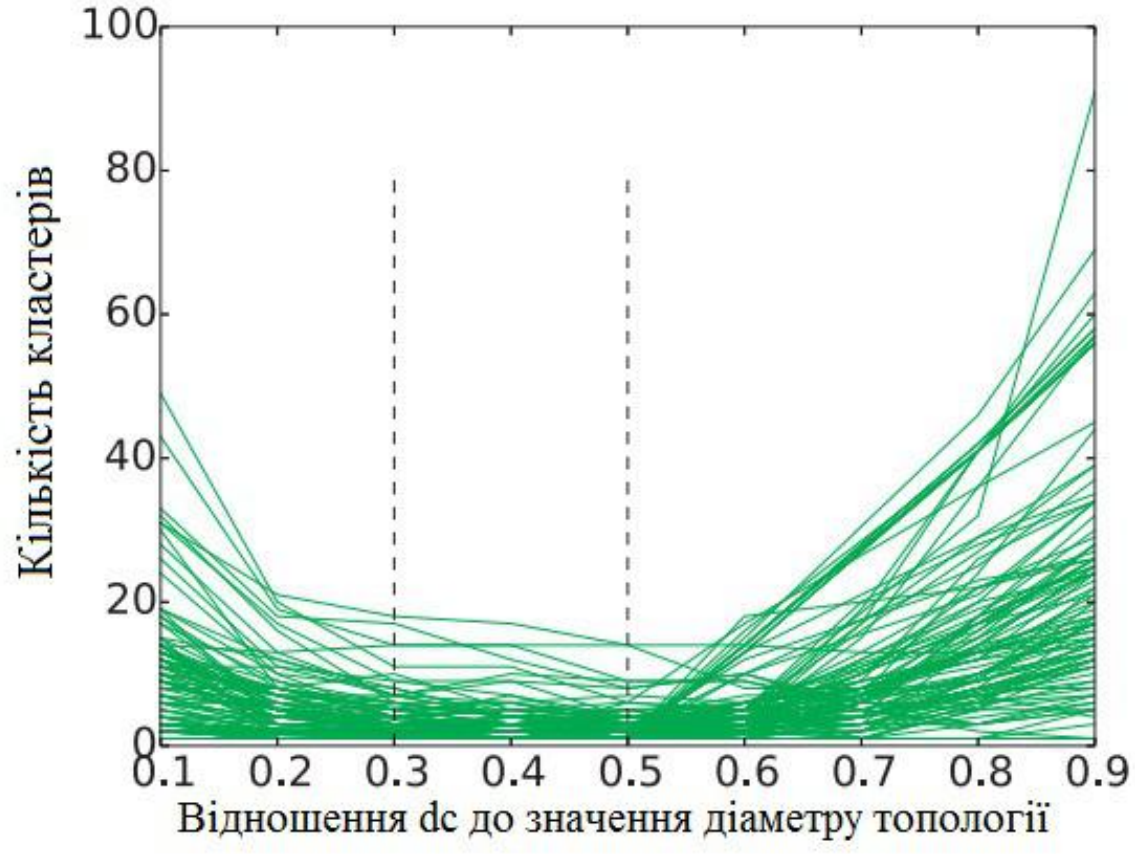


Рис. 4.5 Графік залежності середньої кількості множин від значення граничної відстані множини

Висновки до четвертого розділу

Програма перевірена.

Результати роботи проаналізовано та порівняно з існуючими алгоритмами. Розроблений алгоритм є найкращим з точки зору параметрів продуктивності, розподілу сервісного трафіку по групах та показників топологічної насиченості при розміщенні контролерів.

Проаналізовано вплив параметра максимального значення найближчої дистанції до маршрутизатора на роботу алгоритму.

ВИСНОВКИ

У дипломній роботі розроблено метод кластеризації мобільної мережі. Цей алгоритм також забезпечує рівний розподіл навантажень до контролерів. Демонструє високу ефективність з точки зору службового трафіку та затримки швидкості.

Основними науковими та практичними результатами дослідження є:

1. В ході роботи було сформовано математичну модель проблеми кластеризації мережі SDN та встановлено основні принципи заданого методу; розглядаються існуючі множинні методи. На цій основі був розроблений метод кластеризації мобільної мережі
2. Також розглядається проблема розміщення контролера в підмережі, і для використання в розробленому алгоритмі вибрано рішення.
3. Таким чином, при дослідженні було створено алгоритм, який міг би вирішити проблему розташування контролера у два етапи: мережева кластеризація та вибір місця управління у кожній підмережі. Особливості алгоритму пояснено на прикладі.
4. Створена програмна модель алгоритму, розроблена з використанням мови Java.
5. Висновок, розроблений в результаті тестування програмної моделі на різних мережевих топологіях та аналізу отриманих результатів, показує те, що метод демонструє високу ефективність порівняно з існуючими показниками затримки та швидкості руху трафіку.

З практичної точки зору результати, отримані в цій роботі, дозволяють підвищити ефективність управління мережами SDN, особливо масштабними мобільними мережами.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Sezer, S. Are we ready for SDN? Implementation challenges for software-defined networks / Sezer, S., Scott-Hayward, S., Chouhan, P.K., Fraser, B., Lake, D., Finnegan, J., Viljoen, N., Miller, M., Rao, N. // Communications Magazine, IEEE – 2013 – № 51(7) – С. 36–43.

2. Hakiri, A. Software-defined networking: Challenges and research opportunities for future internet. // Computer Networks, v. 75 – 2014 – С. 453-471.

3. Hu, F. A survey on software-defined network and openflow: From concept to implementation / Hu, F .; Hao, Q .; Bao, K. // IEEE Communications Surveys & Tutorials, v. 16 – 2014 – С. 2181-2206.

4. McKeown, N. OpenFlow: Enabling Innovation in Campus Networks / N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner // SIGCOMM CCR – 2008 – № 38(2).

5. Heller, B. The Controller Placement Problem / B. Heller, R. Sherwood, and N. McKeown // HotSDN – 2012.

6. Liao, J. Density cluster based approach for controller placement problem in large-scale software defined networkings / Jianxin Liao, Haifeng Sun, Jingyu Wang, Qi Qi, Kai Li, Tonghong Li // Computer Networks – 2017 – № 112 – С. 24–35.

7. Hock, D. Pareto-Optimal Resilient Controller Placement in SDN-based Core Networks / David Hock, Matthias Hartmann, Steffen Gebert, Jarschel, Thomas Zinner, Phuoc Tran-Gia // 25th International Teletraffic Congress (ITC) – 2013.

8. Preparata, F. P. Computational Geometry: An Introduction / Franco P. Preparata, Michael Shamos // Springer New York – 1993 – 398 с.

9. Yao, G. On the capacitated controller placement problem in software defined networks / G. Yao , J. Bi , Y. Li , L. Guo // IEEE Commun. Lett. – 2014 – № 18 (8) – С. 1339–1342.

10. Tootoonchian, A. Hyperflow: A distributed control plane for openflow / Tootoonchian, A., Ganjali, Y. // Proceedings of the 2010 internet network management conference on Research on enterprise networking – 2010– С. 3.

11. Жданова, Е. Г. Математическая модель задачи разбиения сети на зоны маршрутизации / Е. Г. Жданова, А.В. Коган, Ю.А. Кулаков, М.О. Сперкач // Математичне та імітаційне моделювання систем: Тринадцята міжнар. наук.-практ. конф., 25 - 29 червня 2018 р.: . тез. доп. – Чернігів, 2018. – С 252-254.

12. Chaudhuri, S. The p-Neighbor k-Center Problem / S. Chaudhuri, N. Garg, and R. Ravi // IPL, vol. 65, no. 3 – 1998.

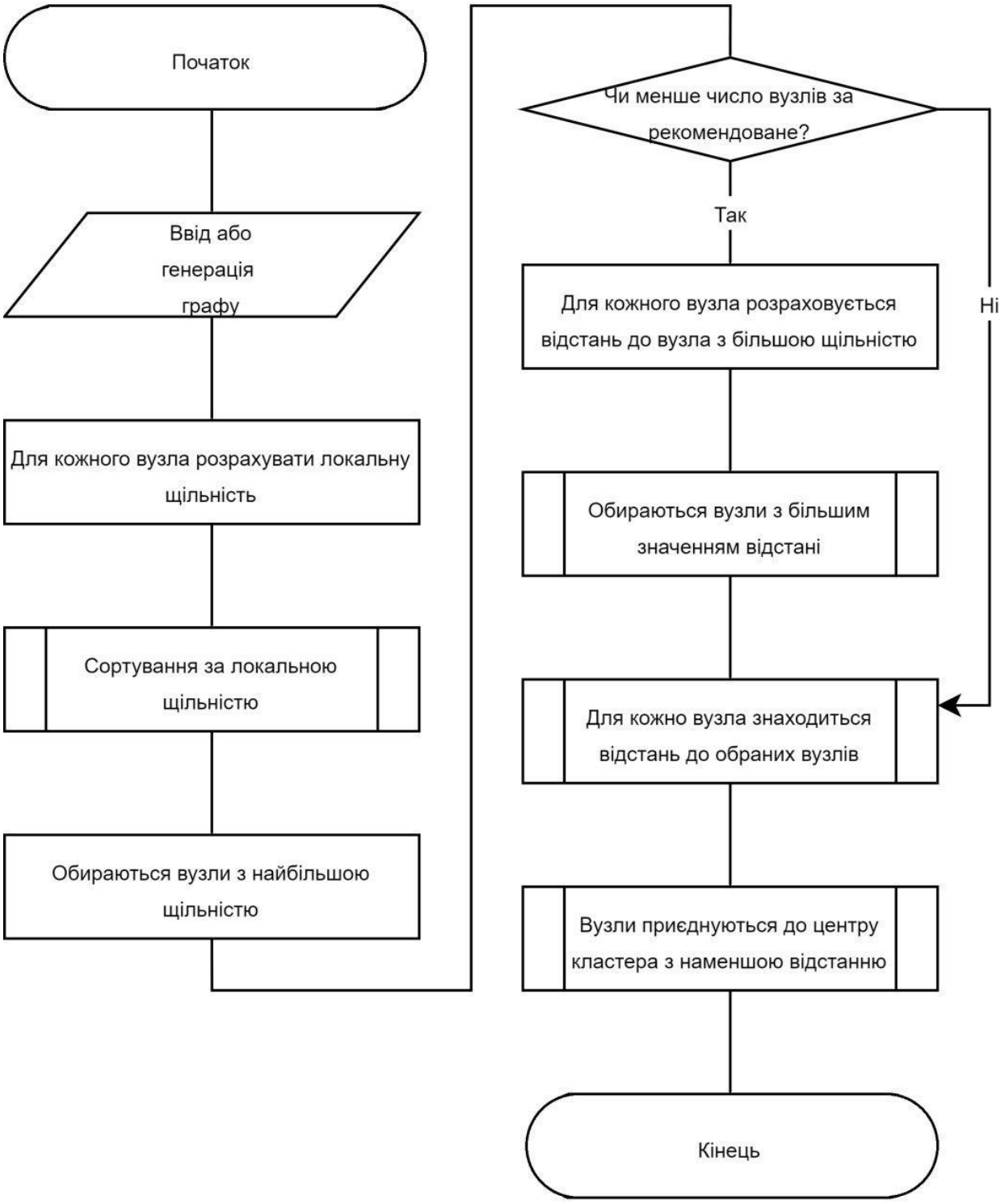
13. Clauset, A. Finding community structure in very large networks / A. Clauset , M.E. Newman , C. Moore // Phys. Rev. – – № 70 (6) – С. 264–277.

14. Dixit, A. Towards an elastic distributed SDN controller / A. Dixit, F. Hao, S. Mukherjee, T.V. Lakshman , R. Kompella // Acm Sigcomm Comput. Commun. Rev. – 2013 – № 43 (4) – С. 7–12.

15. Guo, M. Controller placement for improving resilience of software-defined networks / M. Guo, P. Bhattacharya // International Conference on Networking and Distributed Computing – 2013 – С. 23–27.

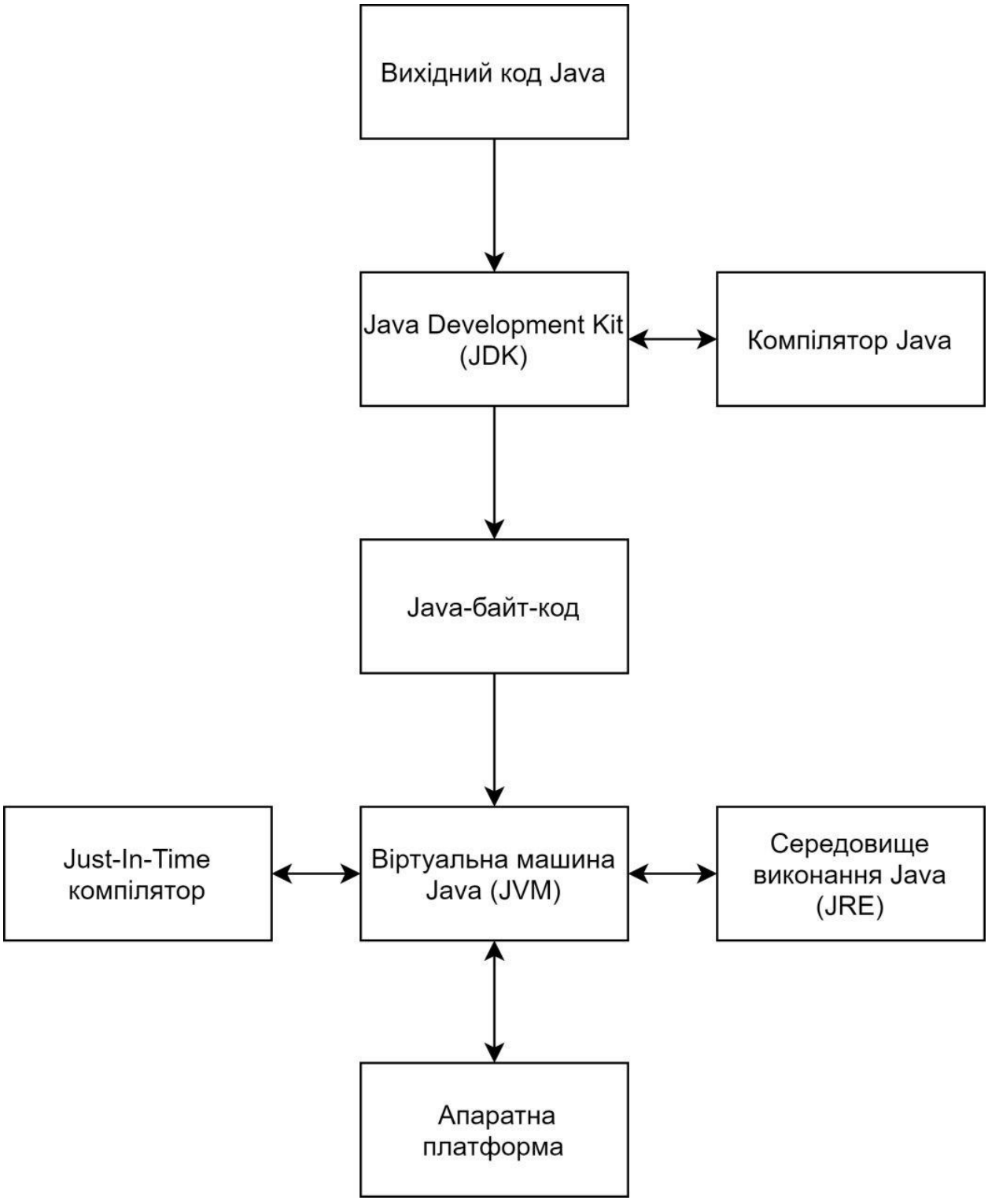
ДОДАТКИ

Схема Алгоритму



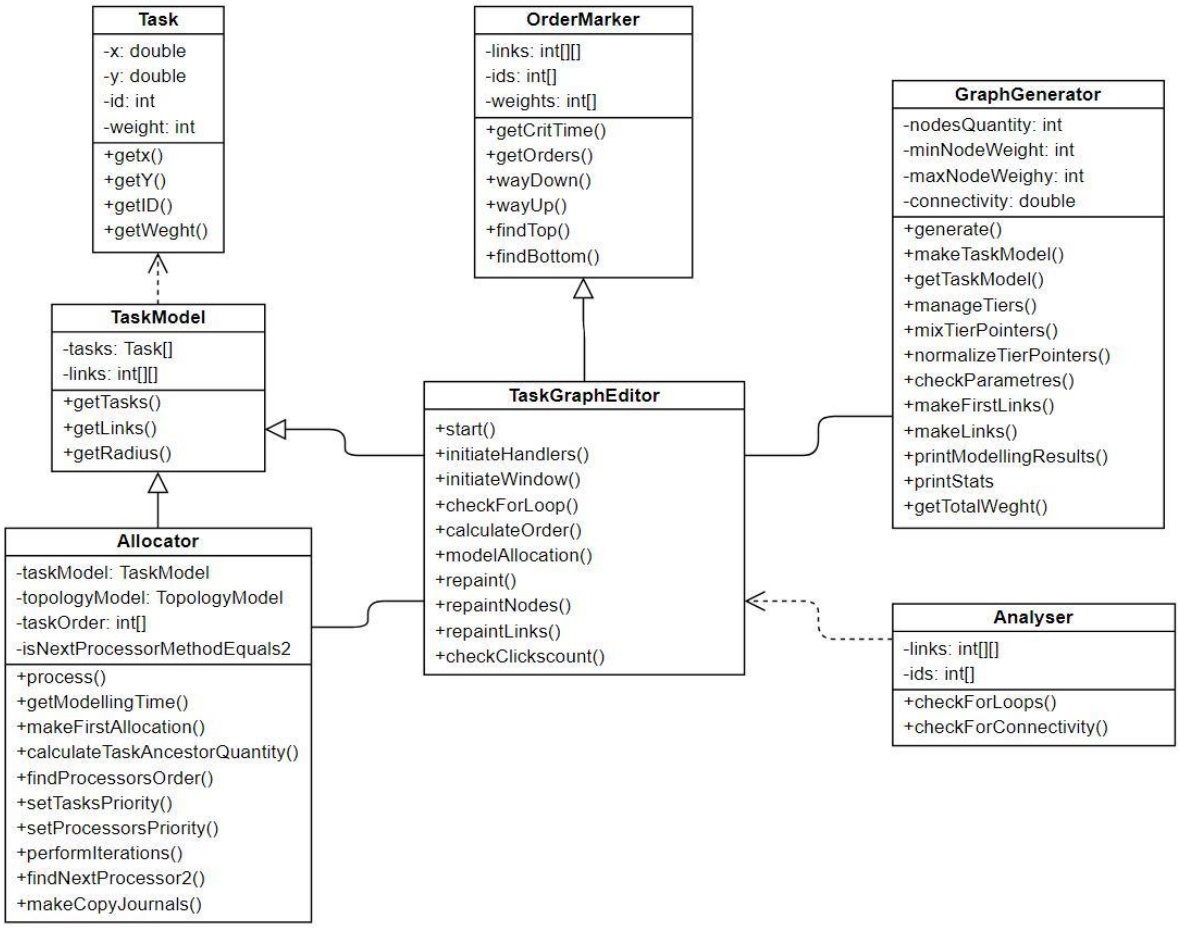
					ДП.6404.04.000 Д1		
Зм.	**Арк.**	*№ документа*	*Підп.*	*Дата*			
Розробив		*Василенко О. В.*			*Спосіб кластеризації мобільних SDN мереж*	*Літ.*	*Аркуш* **Аркушів**
Перевір.		*Калюжний О.О.*				Т	1 1
Н.конт		*Сімоненко В.П.*			*Додаток А*	*НТУУ «КПІ імені Ігоря Сікорського», ФІОТ Група ІО - 64*	
Затв.							

Функціональна Схема



					ДП.6404.05.000 Д2			
Зм.	**Арк.**	*№ документа*	*Підп.*	*Дата*				
					Спосіб кластеризації мобільних SDN мереж	*Літ.*	*Аркуш*	**Аркушів**
Перевірив		*Калюжний О.О.*				Т	1	1
Розробив		*Василенко О.В.*				*НТУУ «КПІ імені Ігоря Сікорського», ФІОТ Група ІО - 64*		
Н.конт		*Сімоненко В.П.*						
Затв.					*Додаток Б*			

Діграма Класів



					ДП.6404.06.000 Д3		
Зм.	Арк.	№ документа	Підп.	Дата			
Розробив		Василенко О. В.			Спосіб кластеризації мобільних SDN мереж Додаток В	Літ.	Аркуш 1 / шв 1
Перевірив		Калюжний О.О.				Т	
Н.конт		Симоненко В.П.				НТУУ «КПІ імені Ігоря Сікорського», ФІОТ Група ІО - 64	
Затв.							

Лістінг Програми

Клас Task

```
package pzks.model;

import java.io.Serializable;

public class Task implements Serializable {
    double x;
    double y;
    int id;
    int weight;

    public Task(double x, double y, int weight, int id) {
        this.x = x;
        this.y = y;
        this.id=id;
        this.weight = weight;
    }

    public double getX() {
        return x;
    }

    public void setX(double x) {
        this.x = x;
    }

    public double getY() {
        return y;
    }

    public void setY(double y) {
        this.y = y;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public int getWeight() {
        return weight;
    }

    public void setWeight(int weight) {
        this.weight = weight;
    }
}
```

Клас Task Model

```
package pzks.model;

import java.io.Serializable;
import java.util.ArrayList;
public class TaskModel implements Serializable{

    ArrayList<Task> tasks;
    int[][] links;
    int radius = 20;
    int maxId = 0;

    public TaskModel() {
        links = new int[0][0];
        tasks = new ArrayList<>();
    }
```

```
public void setTasks(int[] tasksWeights) {
    tasks = new ArrayList<>();
    for(int i=0; i<tasksWeights.length; i++){
        tasks.add(new Task(0,0, tasksWeights[i], i+1));
    }
}

public void setLinks(int[][] newLinks){
    links = new int[newLinks.length][newLinks.length];
    for(int i=0; i<links.length; i++){
        for(int j=0; j<links.length; j++){
            links[i][j] = newLinks[i][j];
        }
    }
}

public ArrayList<Task> getTasks() {
    return tasks;
}

public int[][] getLinks() {
    return links;
}

public int getRadius() {
    return radius;
}

int findByCoordinates(double x, double y) {
    int collLength = tasks.size();
    for (int i = 0; i < collLength; i++) {
        Task task = tasks.get(i);
        double x0 = task.getX();
        double y0 = task.getY();
        double delta = (x - x0) * (x - x0) + (y - y0) * (y - y0);
        if (delta <= radius * radius) {
            return i;
        }
    }
    return -1;
}

public void addNode(double[] clickData) {
    tasks.add(new Task(clickData[0], clickData[1], (int) clickData[4],++maxId));
    int[][] newLinks = new int[links.length + 1][links.length + 1];
    for (int i = 0; i < links.length; i++) {
        System.arraycopy(links[i], 0, newLinks[i], 0, links.length);
    }
    links = newLinks;
}

public void deleteNode(double[] clickData) {

    double x = clickData[0];
    double y = clickData[1];
    int id = findByCoordinates(x, y);
    if (id != -1) {
        int[][] newLinks = new int[links.length - 1][links.length - 1];
        for (int i = 0; i < id; i++) {
            System.arraycopy(links[i], 0, newLinks[i], 0, id);
            System.arraycopy(links[i], id+1, newLinks[i], id, links.length - id - 1);

        }
        for (int i = id + 1; i < links.length; i++) {
            System.arraycopy(links[i], 0, newLinks[i-1], 0, id);
            System.arraycopy(links[i], id+1, newLinks[i-1], id, links.length - id - 1);
        }
        links = newLinks;
        tasks.remove(id);
    }
}

public void changeNodeWeight(double[] clickData) {
    double x = clickData[0];
    double y = clickData[1];
    int id = findByCoordinates(x, y);
    if (id != -1) {
        tasks.get(id).setWeight((int)clickData[4]);
    }
}
```

```
    public void changeNodeWeight(double[] clickData) {
        double x = clickData[0];
        double y = clickData[1];
        int id = findByCoordinates(x, y);
        if (id != -1) {
            tasks.get(id).setWeight((int)clickData[4]);
        }
    }

    public void addLink(double[] clickData) {
        double x1 = clickData[0];
        double y1 = clickData[1];
        int id1 = findByCoordinates(x1, y1);
        double x2 = clickData[2];
        double y2 = clickData[3];
        int id2 = findByCoordinates(x2, y2);
        if((id1!=-1&&id2!=-1)&&(id1!=id2)){
            links[id1][id2]=(int)clickData[4];
            links[id2][id1]=0;
        }
    }

    public void deleteLink(double[] clickData) {
        double x1 = clickData[0];
        double y1 = clickData[1];
        int id1 = findByCoordinates(x1, y1);
        double x2 = clickData[2];
        double y2 = clickData[3];
        int id2 = findByCoordinates(x2, y2);
        if(id1!=-1&&id2!=-1){
            links[id1][id2]=0;
        }
    }

    public void changeLinkWeight(double[] clickData) {
        addLink(clickData);
    }

}
```

Клас TaskGraphEditor

```
package pzks.model;

import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextArea;
import javafx.scene.input.MouseEvent;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Line;
import javafx.scene.shape.Polygon;
import javafx.scene.shape.Rectangle;
import javafx.scene.text.Font;
import javafx.stage.FileChooser;
import javafx.stage.Stage;

import java.io.*;

public class TaskGraphEditor extends Application {
    Button saveTaskGraphButton;
    Button openTaskGraphButton;
    Button loopCheckButton;
    Button calculateButton;
    Button openTopologyGraphButton;
    Button modelButton;

    Button[] editGraphButtons;
    Group buttonsGroup;
    Group root;
    Group shapes;
    String[] buttonNames = {"Add node", "Delete node", "Change weight of node", "Move node",
            "Add link", "Delete link", "Change weight of link"
    };
    Stage stage;
    int[] buttonClicks = {1, 1, 1, 2,
            2, 2, 2};
    int editMode = -1;
    int clickCount = 0;
    int maxCount = 0;
    int radius;
    double arrowAngle = Math.PI / 8;
    double arrowSize = 15;
    double[] clickData;
```

```
Label weightLabel = new Label("Input weight:");
TextArea weightArea = new TextArea();
TextArea logArea = new TextArea();
Font buttonFont = Font.font("SanSerif", 12);
Rectangle backRectangle;
int[][] orders;

private EventHandler<MouseEvent> nodeClickHandler;
private EventHandler<MouseEvent> buttonClickHandler;
TaskModel taskModel;
TopologyModel topologyModel;


@Override
public void start(Stage primaryStage) throws Exception {
   root = new Group();
   stage = primaryStage;
   initiateWindow();
   primaryStage.setTitle("Task graph editor");
   primaryStage.setScene(new Scene(root, 300, 675));
   primaryStage.show();
}

public void initiateHandlers() {
   nodeClickHandler = new EventHandler<MouseEvent>() {
      @Override
      public void handle(MouseEvent e) {

         if (editMode != -1) {
            clickCount++;

            if (clickCount == 1) {
               clickData[0] = e.getSceneX();
               clickData[1] = e.getSceneY();
            } else {
               clickData[2] = e.getSceneX();
               clickData[3] = e.getSceneY();
            }


            String s = "Rec area " + e.getSceneX() + " " + e.getSceneY();
            logArea.setText(s);

            checkClicksCount();
         }


      }
   };

   buttonClickHandler = new EventHandler<MouseEvent>() {
      @Override
      public void handle(MouseEvent e) {
         String s = "" + e.getSceneX() + " " + e.getSceneY();

         Button b = (Button) e.getSource();
         s = b.getText();

         clickCount = 0;

         for (int i = 0; i < buttonNames.length; i++) {
            if (buttonNames[i].equals(s)) {
               editMode = i;
               clickCount = 0;
               maxCount = buttonClicks[i];
               s = "edit=" + editMode + " clickCount=" + clickCount +
                     " maxCount=" + maxCount;
               break;
            }
         }


         logArea.setText(s);

      }
   };

}

public void initiateWindow() {
   initiateHandlers();
   buttonsGroup = new Group();
   editGraphButtons = new Button[buttonNames.length];
   for (int i = 0; i < buttonNames.length; i++) {
      editGraphButtons[i] = new Button();
      editGraphButtons[i].setLayoutX(10);
      editGraphButtons[i].setLayoutY(60 * i);
      editGraphButtons[i].setPrefSize(100, 50);
      editGraphButtons[i].setWrapText(true);
```

```
    editGraphButtons[i].setText(buttonNames[i]);
    editGraphButtons[i].setFont(buttonFont);
    editGraphButtons[i].addEventFilter(MouseEvent.MOUSE_CLICKED, buttonClickHandler);
    buttonsGroup.getChildren().add(editGraphButtons[i]);
}

saveTaskGraphButton = new Button();
saveTaskGraphButton.setLayoutX(130);
saveTaskGraphButton.setLayoutY(0);
saveTaskGraphButton.setPrefSize(100, 50);
saveTaskGraphButton.setWrapText(true);
saveTaskGraphButton.setText("Save to file");
saveTaskGraphButton.setFont(buttonFont);
buttonsGroup.getChildren().add(saveTaskGraphButton);
final FileChooser fileChooser = new FileChooser();
fileChooser.getExtensionFilters().addAll(
        new FileChooser.ExtensionFilter("All files", "*.*"),
        //new FileChooser.ExtensionFilter("JPG", "*.jpg"),
        new FileChooser.ExtensionFilter("TaskModel files", "*.tas")
);
saveTaskGraphButton.setOnAction(
        (final ActionEvent e) -> {
            File file = fileChooser.showSaveDialog(stage);
            if (file != null) {
                saveFile(file);
            }
        });

openTaskGraphButton = new Button();
openTaskGraphButton.setLayoutX(130);
openTaskGraphButton.setLayoutY(60);
openTaskGraphButton.setPrefSize(100, 50);
openTaskGraphButton.setWrapText(true);
openTaskGraphButton.setText("Open file");
openTaskGraphButton.setFont(buttonFont);
buttonsGroup.getChildren().add(openTaskGraphButton);
final FileChooser fileChooser1 = new FileChooser();
fileChooser1.getExtensionFilters().addAll(
        new FileChooser.ExtensionFilter("All files", "*.*"),
        //new FileChooser.ExtensionFilter("JPG", "*.jpg"),
        new FileChooser.ExtensionFilter("TaskModel files", "*.tas")
);
openTaskGraphButton.setOnAction(
        (final ActionEvent e) -> {
            File file = fileChooser1.showOpenDialog(stage);
            if (file != null) {
                openTaskModelFile(file);
            }
        });

loopCheckButton = new Button();
loopCheckButton.setLayoutX(130);
loopCheckButton.setLayoutY(180);
loopCheckButton.setPrefSize(100, 50);
loopCheckButton.setWrapText(true);
loopCheckButton.setText("Check for loop");
loopCheckButton.setFont(buttonFont);
buttonsGroup.getChildren().add(loopCheckButton);
loopCheckButton.setOnAction(
        (final ActionEvent e) -> {
            checkForLoop();
        });

calculateButton = new Button();
calculateButton.setLayoutX(130);
calculateButton.setLayoutY(240);
calculateButton.setPrefSize(100, 50);
calculateButton.setWrapText(true);
calculateButton.setText("Find order of the tasks");
calculateButton.setFont(buttonFont);
buttonsGroup.getChildren().add(calculateButton);
calculateButton.setOnAction(
        (final ActionEvent e) -> {
            calculateOrder();
        });

openTopologyGraphButton = new Button();
openTopologyGraphButton.setLayoutX(130);
openTopologyGraphButton.setLayoutY(360);
openTopologyGraphButton.setPrefSize(100, 50);
openTopologyGraphButton.setWrapText(true);
openTopologyGraphButton.setText("Open file with system topology");
openTopologyGraphButton.setFont(buttonFont);
buttonsGroup.getChildren().add(openTopologyGraphButton);
final FileChooser fileChooser2 = new FileChooser();
fileChooser2.getExtensionFilters().addAll(
        new FileChooser.ExtensionFilter("All files", "*.*"),
        //new FileChooser.ExtensionFilter("JPG", "*.jpg"),
        new FileChooser.ExtensionFilter("TopologyModel files", "*.top")
```

```
        );
        openTopologyGraphButton.setOnAction(
                (final ActionEvent e) -> {
                    File file = fileChooser2.showOpenDialog(stage);
                    if (file != null) {
                        openTopologyModelFile(file);
                    }
                });

        modelButton = new Button();
        modelButton.setLayoutX(130);
        modelButton.setLayoutY(420);
        modelButton.setPrefSize(100, 50);
        modelButton.setWrapText(true);
        modelButton.setText("Model allocation");
        modelButton.setFont(buttonFont);
        buttonsGroup.getChildren().add(modelButton);
        modelButton.setOnAction(
                (final ActionEvent e) -> {
                    modelAllocation();
                });


        weightLabel.setLayoutX(10);
        weightLabel.setLayoutY(420);
        weightLabel.setPrefSize(100, 50);
        weightLabel.setFont(buttonFont);
        buttonsGroup.getChildren().add(weightLabel);

        weightArea.setLayoutX(10);
        weightArea.setLayoutY(470);
        weightArea.setPrefSize(100, 30);
        weightArea.setText("1");
        weightArea.setFont(buttonFont);
        buttonsGroup.getChildren().add(weightArea);

        logArea.setLayoutX(10);
        logArea.setLayoutY(550);
        logArea.setPrefSize(250, 200);
        logArea.setText("Log area");
        logArea.setFont(buttonFont);
        buttonsGroup.getChildren().add(logArea);

        root.getChildren().add(buttonsGroup);


        taskModel = new TaskModel();
        radius = taskModel.getRadius();
        repaint();

    }

    void checkForLoop() {
        int[] taskIds = new int[taskModel.getTasks().size()];
        for (int i = 0; i < taskModel.getTasks().size(); i++) {
            taskIds[i] = taskModel.getTasks().get(i).getId();
        }
        Analyser analyser = new Analyser(taskModel.getLinks(), taskIds);
        String result = analyser.checkForLoops();
        logArea.setText(result);

    }

    void calculateOrder() {
        int[] taskIds = new int[taskModel.getTasks().size()];
        int[] weights = new int[taskModel.getTasks().size()];
        for (int i = 0; i < taskModel.getTasks().size(); i++) {
            taskIds[i] = taskModel.getTasks().get(i).getId();
            weights[i] = taskModel.getTasks().get(i).getWeight();
        }
        OrderMaker analyser = new OrderMaker(taskModel.getLinks(), taskIds, weights);
        orders = analyser.getOrders();
        //String result = analyser.checkForLoops();
        //logArea.setText(result);

    }

    void modelAllocation() {

        String text = "allocation" + '\n' + "order variant: ";
        int mode = readFromTextArea();

        int orderType = mode / 10 - 1;
        switch(orderType){
            case 0: {
                text+="2 laba";
            }
            break;
            case 1: {
```

```
            text+="3 laba";
        }
        break;
        case 2: {
            text+="4 laba";
        }
        break;
    }
    text+= '\n' + "allocation variant: ";
    int allocatorType = mode % 10 - 1;
    boolean is6Laba = true;
    switch(allocatorType){
        case 0: {
            text+="6 laba";
            is6Laba=true;
        }
        break;
        case 1: {
            text+="7 laba";
            is6Laba=false;
        }
        break;
    }
    logArea.setText(text);
    System.out.println("");
    System.out.println(text);
    Allocator allocator = new Allocator(taskModel, topologyModel, orders[orderType], is6Laba);
    allocator.process();


}


void openTaskModelFile(File file) {
    try {
        FileInputStream fi = new FileInputStream(file);
        ObjectInputStream oi = new ObjectInputStream(fi);

        // Read objects
        taskModel = (TaskModel) oi.readObject();
        repaint();

        oi.close();
        fi.close();


    } catch (FileNotFoundException e) {
        System.out.println("File not found");
    } catch (IOException e) {
        System.out.println("Error initializing stream");
    } catch (ClassNotFoundException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}

void openTopologyModelFile(File file) {
    try {
        FileInputStream fi = new FileInputStream(file);
        ObjectInputStream oi = new ObjectInputStream(fi);

        // Read objects
        topologyModel = (TopologyModel) oi.readObject();
        repaint();

        oi.close();
        fi.close();


        logArea.setText(file.getName() + " is opened");

    } catch (FileNotFoundException e) {
        System.out.println("File not found");
    } catch (IOException e) {
        System.out.println("Error initializing stream");
    } catch (ClassNotFoundException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}

void saveFile(File file) {
    try {
        FileOutputStream f = new FileOutputStream(file);
        ObjectOutputStream o = new ObjectOutputStream(f);

        // Write objects to file
        o.writeObject(taskModel);
```

```
            o.close();
            f.close();


        } catch (FileNotFoundException e) {
            System.out.println("File not found");
        } catch (IOException e) {
            System.out.println("Error initializing stream");
        }
    }

    void repaint() {
        root.getChildren().remove(shapes);
        backRectangle = new Rectangle();
        backRectangle.setX(300);
        backRectangle.setY(0);
        backRectangle.setFill(Color.WHITESMOKE);
        backRectangle.setStroke(Color.WHITE);
        backRectangle.setWidth(1000);
        backRectangle.setHeight(800);
        backRectangle.addEventFilter(MouseEvent.MOUSE_CLICKED, nodeClickHandler);

        clickData = new double[5];

        shapes = new Group();
        shapes.getChildren().add(backRectangle);
        shapes.getChildren().add(repaintNodes());
        shapes.getChildren().add(repaintLinks());
        root.getChildren().add(shapes);

    }

    Group repaintNodes() {
        Group tempGroup = new Group();
        for (int i = 0; i < taskModel.getTasks().size(); i++) {
            Task task = taskModel.getTasks().get(i);

            Circle tempCircle = new Circle();
            tempCircle.setRadius(radius);
            tempCircle.setCenterX(task.getX());
            tempCircle.setCenterY(task.getY());
            tempCircle.addEventFilter(MouseEvent.MOUSE_CLICKED, nodeClickHandler);

            tempCircle.setFill(Color.WHITE);
            tempCircle.setStroke(Color.GREEN);

            tempGroup.getChildren().add(tempCircle);

            int taskId = task.getId();
            int taskWeight = task.getWeight();
            String taskName = taskId + "[" + taskWeight + "]";
            Label tempLabel = new Label();
            tempLabel.setText(taskName);
            tempLabel.setLayoutX(task.getX() - 10);
            tempLabel.setLayoutY(task.getY() - 7);
            tempLabel.setFont(buttonFont);
            tempGroup.getChildren().add(tempLabel);


        }
        return tempGroup;
    }

    Group repaintLinks() {
        Group tempGroup = new Group();
        int[][] links = taskModel.getLinks();
        for (int i = 0; i < taskModel.getTasks().size(); i++) {
            for (int j = 0; j < taskModel.getTasks().size(); j++) {
                if (links[i][j] != 0) {
                    Task task1 = taskModel.getTasks().get(i);
                    Task task2 = taskModel.getTasks().get(j);
                    double[] points = new double[4];
                    points[0] = task1.getX();
                    points[1] = task1.getY();
                    points[2] = task2.getX();
                    points[3] = task2.getY();
                    double[] newPoints = calculatePoints(points);
                    Line line = new Line(newPoints[0], newPoints[1], newPoints[2], newPoints[3]);
                    line.setStrokeWidth(2);
                    line.setStroke(Color.BLACK);
                    tempGroup.getChildren().add(line);

                    Label label = new Label();
                    label.setText("[" + links[i][j] + "]");
                    label.setLayoutX((newPoints[0] + newPoints[2]) / 2);
                    label.setLayoutY((newPoints[1] + newPoints[3]) / 2 - 30);
                    label.setFont(buttonFont);
                    tempGroup.getChildren().add(label);
```

```
                Polygon polygon = new Polygon();
                polygon.getPoints().addAll(new Double[]{
                        newPoints[2], newPoints[3],
                        newPoints[2] + arrowSize * Math.sin(newPoints[5] - arrowAngle), newPoints[3] + arrowSize * Math.cos(newPoints[5] - arrowAngle),
                        newPoints[2] + arrowSize * 2 / 3 * Math.sin(newPoints[5]), newPoints[3] + arrowSize * 2 / 3 * Math.cos(newPoints[5]),
                        newPoints[2] + arrowSize * Math.sin(newPoints[5] + arrowAngle), newPoints[3] + arrowSize * Math.cos(newPoints[5] + arrowAngle)});
                polygon.setFill(Color.BLACK);

                tempGroup.getChildren().add(polygon);


            }
        }
    }
    return tempGroup;
}

double[] calculatePoints(double[] points) {
    double[] newPoints = new double[6];
    double deltaX = points[2] - points[0];
    double deltaY = points[3] - points[1];
    double squares = Math.pow(points[2] - points[0], 2) + Math.pow(points[3] - points[1], 2);
    squares = Math.pow(squares, 0.5);
    double angle = Math.atan(deltaX / deltaY);
    if (deltaY < 0)
        angle += Math.PI;
    for (int i = 0; i < 4; i++) {
        newPoints[i] = points[i];
    }
    double angle1 = angle + Math.PI;
    newPoints[0] += radius * Math.sin(angle);
    newPoints[1] += radius * Math.cos(angle);
    newPoints[2] += radius * Math.sin(angle1);
    newPoints[3] += radius * Math.cos(angle1);
    newPoints[4] = angle;
    newPoints[5] = angle1;
    return newPoints;
}


void checkClicksCount() {
    if (maxCount == clickCount) {

        switch (editMode) {
            case 0: {
                clickData[4] = readFromTextArea();
                taskModel.addNode(clickData);
                logArea.setText("Add node");
            }
            break;

            case 1: {
                taskModel.deleteNode(clickData);
                logArea.setText("Delete node");
            }
            break;

            case 2: {
                clickData[4] = readFromTextArea();
                taskModel.changeNodeWeight(clickData);
                logArea.setText("Change weight of node");
            }
            break;


            case 3: {
                taskModel.moveNode(clickData);
                logArea.setText("Move node");
            }
            break;


            case 4: {
                clickData[4] = readFromTextArea();
                taskModel.addLink(clickData);
                logArea.setText("Add link");
            }
            break;


            case 5: {
                taskModel.deleteLink(clickData);
                logArea.setText("Delete link");
            }
            break;


            case 6: {
                clickData[4] = readFromTextArea();
```

```
                taskModel.changeLinkWeight(clickData);
                logArea.setText("Change weight of link");
            }
            break;
        }
        repaint();
        editMode = -1;
        clickCount = 0;
        clickData = new double[5];
    }
  }

  int readFromTextArea() {
    //System.out.println(weightArea.getText());
    return Integer.parseInt(weightArea.getText());
  }

  public static void main(String[] args) {
    launch(args);
  }
}
```

Клас Analyser

```
package pzks.model;

import java.util.LinkedList;

public class Analyser {

  int[][] links;
  int[] colors;
  int[] lastNodes;
  int[] ids;
  String status;


  public Analyser(int[][] links, int[] ids) {
    this.links = new int[links.length][links.length];
    this.ids = new int[links.length];
    colors = new int[links.length];
    lastNodes = new int[links.length];
    for(int i=0; i<links.length; i++) {
      this.ids[i] = ids[i];
      for(int j=0; j<links.length; j++) {
        this.links[i][j] = links[i][j];
      }
    }

    status = "";
  }

  String checkForLoops(){
    for(int i=0; i<colors.length&&status.equals(""); i++){
      colors = new int[links.length];
      lastNodes = new int[links.length];
      checkVertexForLoop(i);
    }

    if(status.equals("")){
      status = "everything is Ok";
    }

    return status;

  }

  void checkVertexForLoop(int id) {
    for(int i=0; i<colors.length&&status.equals(""); i++){
      if((i!=id)&&(links[id][i]!=0)&&status.equals("")){
        switch(colors[i]) {
          case 0: { //white
            colors[i]=1;
            lastNodes[i]=id;
            checkVertexForLoop(i);
          }
          break;

          case 1: { //grey

            lastNodes[i]=id;
            status+=ids[id];
            findBackWay(lastNodes[id], id);
          }
          break;


          case 2: { // black
```

```
                }
                break;

            }
        }
    }
    colors[id]=2;
}

void findBackWay(int currentId, int finishId){
    if(currentId!=finishId){
        status=ids[currentId]+"-"+status;
        findBackWay(lastNodes[currentId],finishId);
    } else {
        status = "Loop was found: "+ status;
    }

}

String checkForConnectivity() {
    int vertexId = 0;
    for(int i=0; i<colors.length&&status.equals(""); i++){
        colors = new int[links.length];
        lastNodes = new int[links.length];
        checkVertexForConnectivity(vertexId);
    }

    if(status.equals("")){
        status = "everything is Ok";
    } else {
        status = "Node " + ids[vertexId] + " is not connective to nodes: " + status;
    }

    return status;
}

void checkVertexForConnectivity(int id){
    boolean[] checkedNodes = new boolean[links.length];
    LinkedList<Integer> nextWave = new LinkedList<>();
    checkedNodes[id]=true;
    for(int i=0; i<links.length; i++) {
        if(links[id][i]!=0||links[i][id]!=0){
            if(!checkedNodes[i]) {
                checkedNodes[i] = true;
                nextWave.add(i);
            }
        }
    }
    while(!nextWave.isEmpty()){
        int currentId = nextWave.getFirst();
        for(int i=0; i<links.length; i++) {
            if(links[currentId][i]!=0||links[i][currentId]!=0){
                if(!checkedNodes[i]) {
                    checkedNodes[i] = true;
                    nextWave.add(i);
                }
            }
        }
        nextWave.removeFirst();
    }
    for(int i=0; i<links.length; i++){
        if(!checkedNodes[i]){
            if(!status.equals("")){
                status = ids[i] + ", "+ status;
            } else {
                status = Integer.toString(ids[i]);
            }

        }
    }

}
}
```

Клас OrderMaker

```
package pzks.model;

import java.util.Arrays;
import java.util.LinkedList;

public class OrderMaker {

    int[][] links;
    int[] ids;
    int[] weights;
    int[][] parametres;
    boolean[] isMarked;
```

```
int[][] results;

int critTime;

public OrderMaker(int[][] links, int[] ids, int[] weights) {
    this.links = new int[links.length][links.length];
    this.ids = new int[links.length];
    this.weights = new int[links.length];
    for (int i = 0; i < links.length; i++) {
        this.ids[i] = ids[i];
        this.weights[i] = weights[i];
        for (int j = 0; j < links.length; j++) {
            this.links[i][j] = links[i][j];
        }
    }
    parametres = new int[7][links.length];
    wayDown();
    wayUp();

    int tMax = 0;
    for (int i = 0; i < links.length; i++) {
        if (tMax < parametres[0][i] + weights[i])
            tMax = parametres[0][i] + weights[i];
    }

    for (int i = 0; i < links.length; i++) {
        parametres[2][i] = tMax - parametres[1][i];
        parametres[3][i] = parametres[2][i] - parametres[0][i];
    }
    System.arraycopy(weights, 0, parametres[5], 0, links.length);
    System.arraycopy(weights, 0, parametres[6], 0, links.length);

    critTime = 0;
    for(int i=0; i<parametres[1].length; i++)
        if(critTime<parametres[1][i])
            critTime=parametres[1][i];



    int[] generatedArray = new int[links.length];
    for (int i = 0; i < generatedArray.length; i++) {
        generatedArray[i] = i;
    }

    int[][] temp1 = new int[2][links.length];
    System.arraycopy(generatedArray, 0, temp1[0], 0, links.length);
    System.arraycopy(parametres[3], 0, temp1[1], 0, links.length);
    sort1(temp1[0], temp1[1]);

    int[][] temp2 = new int[3][links.length];
    System.arraycopy(generatedArray, 0, temp2[0], 0, links.length);
    System.arraycopy(parametres[4], 0, temp2[1], 0, links.length);
    System.arraycopy(parametres[5], 0, temp2[2], 0, links.length);
    sort2(temp2[0], temp2[1], temp2[2]);

    int[][] temp3 = new int[2][links.length];
    System.arraycopy(generatedArray, 0, temp3[0], 0, links.length);
    System.arraycopy(parametres[6], 0, temp3[1], 0, links.length);
    sort3(temp3[0], temp3[1]);

    results = new int[3][links.length];
    System.arraycopy(temp1[0], 0, results[0], 0, links.length);
    System.arraycopy(temp2[0], 0, results[1], 0, links.length);
    System.arraycopy(temp3[0], 0, results[2], 0, links.length);

}

public int getCritTime() {
    return critTime;
}

void printTranslatedArray(int[] array) {
    if (array.length > 0) {
        String result = "[";
        for (int i = 0; i < array.length - 1; i++) {
            result += ids[array[i]] + ", ";
        }
        result += ids[array[array.length - 1]] + "]";
        System.out.println(result);
    } else {
        System.out.println("[]");
    }

}

int[][] getOrders() {
    return results;
}
```

```
int[] sort1(int[] ids, int[] a) {
    for (int i = 0; i < links.length; i++) {
        for (int j = i + 1; j < links.length; j++) {
            if (a[j] < a[i]) {
                int bufId = ids[i];
                int bufA = a[i];
                ids[i] = ids[j];
                a[i] = a[j];
                ids[j] = bufId;
                a[j] = bufA;
            }
        }
    }
    return ids;
}

int[] sort2(int[] ids, int[] a, int[] b) {
    for (int i = 0; i < links.length; i++) {
        for (int j = i + 1; j < links.length; j++) {
            if (a[j] < a[i] || (a[j] == a[i] && b[j] > b[i])) {
                int bufId = ids[i];
                int bufA = a[i];
                int bufB = b[i];

                ids[i] = ids[j];
                a[i] = a[j];
                b[i] = b[j];

                ids[j] = bufId;
                a[j] = bufA;
                b[j] = bufB;
            }
        }
    }
    return ids;
}

int[] sort3(int[] ids, int[] a) {
    for (int i = 0; i < links.length; i++) {
        for (int j = i + 1; j < links.length; j++) {
            if (a[j] < a[i]) {
                int bufId = ids[i];
                int bufA = a[i];
                ids[i] = ids[j];
                a[i] = a[j];
                ids[j] = bufId;
                a[j] = bufA;
            }
        }
    }
    return ids;
}

void wayDown() {
    LinkedList<Integer> top = findTop();
    isMarked = new boolean[links.length];
    LinkedList<Integer> newWave = new LinkedList<>();
    for (int currentNode : top) {
        isMarked[currentNode] = true;
        for (int i = 0; i < links.length; i++) {
            if (links[currentNode][i] != 0) {
                boolean flag = true;
                for (int j = 0; j < links.length && flag; j++) {
                    if (links[j][i] != 0 && !isMarked[j]) {
                        flag = false;
                    }
                }
                if (flag) {
                    newWave.add(i);
                }
            }
        }
        while (!newWave.isEmpty()) {
            int currentId = newWave.getFirst();

            for (int i = 0; i < links.length; i++) {
                if (links[i][currentId] != 0) {
                    if (parametres[0][i] + weights[i] > parametres[0][currentId]) {
                        parametres[0][currentId] = parametres[0][i] + weights[i];
                    }
                    if (parametres[4][i] + 1 > parametres[4][currentId]) {
                        parametres[4][currentId] = parametres[4][i] + 1;
                    }

                }
            }

            isMarked[currentId] = true;
```

```
            newWave.removeFirst();

            for (int i = 0; i < links.length; i++) {
               if (links[currentId][i] != 0) {
                  boolean flag = true;
                  for (int j = 0; j < links.length && flag; j++) {
                     if (links[j][i] != 0 && !isMarked[j]) {
                        flag = false;
                     }
                  }
                  if (flag) {
                     newWave.add(i);
                  }
               }
            }

         }
      }


   }

   void wayUp() {
      LinkedList<Integer> bottom = findBottom();
      isMarked = new boolean[links.length];
      LinkedList<Integer> newWave = new LinkedList<>();
      for (int currentNode : bottom) {
         isMarked[currentNode] = true;
         parametres[1][currentNode] = weights[currentNode];
         for (int i = 0; i < links.length; i++) {
            if (links[i][currentNode] != 0) {
               boolean flag = true;
               for (int j = 0; j < links.length && flag; j++) {
                  if (links[i][j] != 0 && !isMarked[j]) {
                     flag = false;
                  }
               }
               if (flag) {
                  newWave.add(i);
               }
            }
         }
         while (!newWave.isEmpty()) {
            int currentId = newWave.getFirst();

            for (int i = 0; i < links.length; i++) {
               if (links[currentId][i] != 0) {
                  if (parametres[1][i] + weights[currentId] > parametres[1][currentId]) {
                     parametres[1][currentId] = parametres[1][i] + weights[currentId];
                  }
               }
            }

            isMarked[currentId] = true;
            newWave.removeFirst();

            for (int i = 0; i < links.length; i++) {
               if (links[i][currentId] != 0) {
                  boolean flag = true;
                  for (int j = 0; j < links.length && flag; j++) {
                     if (links[i][j] != 0 && !isMarked[j]) {
                        flag = false;
                     }
                  }
                  if (flag) {
                     newWave.add(i);
                  }
               }
            }

         }
      }


   }


   LinkedList<Integer> findTop() {
      LinkedList<Integer> top = new LinkedList<>();
      for (int i = 0; i < links.length; i++) {
         boolean flag = true;
         for (int j = 0; j < links.length && flag; j++) {
            if (links[j][i] != 0) {
               flag = false;
            }

         }
         if (flag)
            top.add(i);
```

```
        }
        return top;

    }

    LinkedList<Integer> findBottom() {
        LinkedList<Integer> bottom = new LinkedList<>();
        for (int i = 0; i < links.length; i++) {
            boolean flag = true;
            for (int j = 0; j < links.length && flag; j++) {
                if (links[i][j] != 0) {
                    flag = false;
                }

            }
            if (flag)
                bottom.add(i);
        }
        return bottom;

    }
}
```

Клас GraphGenerator

```
package pzks.model;

import java.util.Arrays;
import java.util.LinkedList;

public class GraphGenerator {
    int nodesQuantity;
    int minNodeWeight;
    int maxNodeWeight;
    double connectivity;

    int[] tierSizes;
    int[] tierPointers;
    int[] weights;
    int[][] links;

    double TIER_DISPERSION = 0.3;
    int MIX_NUMBER = 5;
    String status = "";

    double practicalConnectivity;
    int totalWeight;
    int totalLinks;
    int totalLeftLinks;

    TaskModel taskModel;

    public GraphGenerator(int nodesQuantity, int minNodeWeight, int maxNodeWeight, double connectivity) {
        this.nodesQuantity = nodesQuantity;
        this.minNodeWeight = minNodeWeight;
        this.maxNodeWeight = maxNodeWeight;
        this.connectivity = connectivity;
        weights = new int[nodesQuantity];
        links = new int[nodesQuantity][nodesQuantity];
        for (int i = 0; i < nodesQuantity; i++) {
            weights[i] = (int) genNumber(minNodeWeight, maxNodeWeight + 0.999);
        }
    }

    void generate() {


        manageTiers();
        checkParametres();
        int faultNumber = 0;

        if (!status.equals("")) {
            while (!status.equals("") && faultNumber < 100) {
                manageTiers();
                checkParametres();
            }

        }
        if (status.equals("")) {
            makeFirstLinks();
            makeLinks();
            makeTaskModel();
            //System.out.println(1);
            //printModellingResult();
        } else
            System.out.println(status);


    }
```

```
void makeTaskModel(){
    taskModel = new TaskModel();
    taskModel.setLinks(links);
    taskModel.setTasks(weights);
}

public TaskModel getTaskModel() {
    return taskModel;
}

void manageTiers() {
    status = "";
    int tierQuantity = (int) (Math.sqrt(nodesQuantity) * genNumber(1 - TIER_DISPERSION, 1 + TIER_DISPERSION)) + 1;
    createTierPointers(tierQuantity);
    mixTierPointers();
    normalizeTierPointers();
    //System.out.println(Arrays.toString(tierSizes));
    tierPointers = new int[tierSizes.length];
    tierPointers[0] = 0;
    for (int i = 1; i < tierPointers.length; i++) {
        tierPointers[i] = tierPointers[i - 1] + tierSizes[i - 1];
    }


}

void createTierPointers(int n) {
    tierSizes = new int[n];
    int nodesSize = nodesQuantity / n;
    int rest = nodesQuantity % n;
    Arrays.fill(tierSizes, nodesSize);
    for (int i = 0; i < rest; i++) {
        int indexNumber = (int) genNumber(0, tierSizes.length - 0.0001);
        tierSizes[indexNumber]++;
    }


}

void mixTierPointers() {

    int mixLimit = tierSizes.length * MIX_NUMBER;
    for (int i = 0; i < mixLimit; i++) {
        int a = (int) genNumber(0, tierSizes.length - 0.001);
        int b = (int) genNumber(0, tierSizes.length - 0.001);
        if (a != b) {
            double mark = genNumber(0, 1);
            if (mark < 0.5) {
                int c = a;
                a = b;
                b = c;
            }
            int mix = (int) genNumber(0, tierSizes[a] - 0.01);
            if (mix != -1) {
                tierSizes[a] -= mix;
                tierSizes[b] += mix;
            }


        }
    }
}

void normalizeTierPointers() {
    int counter = 0;
    int maxSize = (int) (nodesQuantity / tierSizes.length * 1.5) + 1;
    for (int i = 0; i < tierSizes.length; i++) {
        if (tierSizes[i] > maxSize) {
            counter += tierSizes[i] - maxSize;
            tierSizes[i] = maxSize;
        }
    }
    for (int i = 0; i < counter; i++) {
        int indexNumber = (int) genNumber(0, tierSizes.length - 0.0001);
        tierSizes[indexNumber]++;
    }

}

void checkParametres() {
    totalWeight = 0;
    for (int i = 0; i < nodesQuantity; i++) {
        totalWeight += weights[i];
    }
    totalLeftLinks = (int) (totalWeight * (1 - connectivity) / connectivity);
    totalLinks = totalLeftLinks;
    practicalConnectivity = totalWeight / (totalWeight + totalLeftLinks + .0000000001);
```

```
        int minLeftLinks = 0;
        for (int i = 1; i < tierSizes.length; i++) {
            minLeftLinks += tierSizes[i];
        }
        if (minLeftLinks > totalLeftLinks)
            status = "Decrease connectivity";
    }

    void makeFirstLinks() {
        for (int i = 0; i < tierPointers.length - 1; i++) {
            LinkedList<Integer> prevTier = new LinkedList<>();
            LinkedList<Integer> nextTier = new LinkedList<>();

            LinkedList<Integer> prevTier0 = new LinkedList<>();
            LinkedList<Integer> nextTier0 = new LinkedList<>();
            int a = tierPointers[i];
            int b = tierPointers[i + 1];
            int c = 0;
            if (i + 2 != tierPointers.length) {
                c = tierPointers[i + 2];
            } else {
                c = nodesQuantity;
            }
            for (int j = a; j < b; j++) {
                prevTier.add(j);
            }
            for (int j = b; j < c; j++) {
                nextTier.add(j);
            }
            prevTier0.addAll(prevTier);
            nextTier0.addAll(nextTier);

            if (b - a >= c - b) {
                while (!nextTier.isEmpty()) {

                    int prevIndex = (int) genNumber(0, prevTier.size() - 0.000001);
                    int nextIndex = (int) genNumber(0, nextTier.size() - 0.000001);
                    if (prevIndex < 0)
                        prevIndex = 0;
                    if (nextIndex < 0)
                        nextIndex = 0;

                    totalLeftLinks--;
                    links[prevTier.get(prevIndex)][nextTier.get(nextIndex)] = 1;
                    prevTier.remove(prevIndex);
                    nextTier.remove(nextIndex);
                }
            } else {

                while (!prevTier.isEmpty()) {
                    int prevIndex = (int) genNumber(0, prevTier.size() - 0.000001);
                    int nextIndex = (int) genNumber(0, nextTier.size() - 0.000001);
                    if (prevIndex < 0)
                        prevIndex = 0;
                    if (nextIndex < 0)
                        nextIndex = 0;
                    totalLeftLinks--;
                    links[prevTier.get(prevIndex)][nextTier.get(nextIndex)] = 1;
                    prevTier.remove(prevIndex);
                    nextTier.remove(nextIndex);
                }

                while (!nextTier.isEmpty()) {
                    int prevIndex = (int) genNumber(0, prevTier0.size() - 0.000001);
                    int nextIndex = (int) genNumber(0, nextTier.size() - 0.000001);
                    if (prevIndex < 0)
                        prevIndex = 0;
                    if (nextIndex < 0)
                        nextIndex = 0;
                    totalLeftLinks--;
                    links[prevTier0.get(prevIndex)][nextTier.get(nextIndex)] = 1;
                    nextTier.remove(nextIndex);
                }

            }

        }
    }

    void makeLinks() {
        while (totalLeftLinks != 0) {
            int prevNode = 0;
            if (genNumber(0, 1) < 0.5) {
                prevNode = (int) genNumber(0, tierPointers[tierPointers.length - 1] / 2 - 0.000001);
            } else {
                prevNode = (int) genNumber(0, tierPointers[tierPointers.length - 1] - 0.000001);
            }
            if (prevNode < 0)
                prevNode = 0;
```

```
        LinkedList<Integer> childs = getChildrens(prevNode);
        double mark = 0.85;
        if (childs.size() < 4) {
            mark = 0.75;
        }
        int nextNode = -1;
        if (genNumber(0, 1) < mark) {
            if (childs.size() > 0) {
                nextNode = childs.get((int) genNumber(0, childs.size() - 0.000001));
            } else {
                LinkedList<Integer> notChilds = getNotChildrens(prevNode);
                nextNode = notChilds.get((int) genNumber(0, notChilds.size() - 0.000001));
            }
            //exist
        } else {
            LinkedList<Integer> notChilds = getNotChildrens(prevNode);
            if (notChilds.size() > 0) {
                nextNode = notChilds.get((int) genNumber(0, notChilds.size() - 0.000001));
            } else {
                nextNode = childs.get((int) genNumber(0, childs.size() - 0.000001));
            }
            //notExist
        }
        if(prevNode<nextNode) {
            links[prevNode][nextNode]++;
            totalLeftLinks--;
        }

    }
}

double genNumber(double a, double b) {
    if (a >= b)
        return -1;
    else
        return a + Math.random() * (b - a);
}

double max(double a, double b) {
    if (a > b)
        return a;
    else
        return b;
}


void printModellingResult() {
    printStats();

    System.out.println("links");
    String title = makeSpace(5);
    for (int j = 0; j < nodesQuantity; j++) {
        title += " " + intToString(j + 1);

    }
    System.out.println(title);
    for (int i = 0; i < nodesQuantity; i++) {
        int counter = 0;
        String line = intToString(i + 1) + " : ";
        for (int j = 0; j < nodesQuantity; j++) {

            line += " " + intToString(links[i][j]);
            counter += links[i][j];

        }
        line += "\t\t sum = " + intToString(counter);
        System.out.println(line);
    }

}

void printStats() {
    String[] line = new String[7];
    line[0] = "N=" + nodesQuantity + "; min=" + minNodeWeight + "; max=" + maxNodeWeight + "; con=" + connectivity;
    line[1] = "practCon=" + practicalConnectivity;
    line[2] = "totalWeight=" + totalWeight + "; totalLinks=" + totalLinks;

    line[3] = "weights of nodes:";
    line[4] = makeSpace(5);
    for (int j = 0; j < weights.length; j++) {
        line[4] += " " + intToString(weights[j]);

    }
    line[5] = "tiers pointers:";
    line[6] = makeSpace(5);
    for (int i = 0; i < tierPointers.length; i++) {
        line[6] += " " + intToString(tierPointers[i] + 1);
    }
```

```
        for (int i = 0; i < line.length; i++) {
            System.out.println(line[i]);
        }

    }


    String makeSpace(int n) {
        String result = "";
        for (int i = 0; i < n; i++)
            result += " ";
        return result;
    }

    String intToString(int n) {
        String result = "";
        if (n >= 0 && n < 10)
            result = " ";
        result += n;
        return result;
    }

    LinkedList<Integer> getChildrens(int index) {
        LinkedList<Integer> list = new LinkedList<>();
        for (int i = index; i < nodesQuantity; i++) {
            if (links[index][i] > 0)
                list.add(i);
        }
        return list;
    }

    LinkedList<Integer> getNotChildrens(int index) {
        LinkedList<Integer> list = new LinkedList<>();
        int tier = getTier(index);
        if (tier != tierPointers.length - 1) {
            for (int i = tierPointers[tier + 1]; i < nodesQuantity; i++) {
                if (links[index][i] == 0)
                    list.add(i);
            }
        }
        return list;
    }

    int getTier(int index) {
        int tier = -1;
        if (index > -1 && index < nodesQuantity) {
            tier = 0;
            for (int i = 1; i < tierPointers.length; i++) {
                if (tierPointers[i] <= index) {
                    tier++;
                    break;
                }

            }
        }
        return tier;
    }

    public int getTotalWeight() {
        return totalWeight;
    }
}
```

Клас Allocator

```
package pzks.model;

import java.util.Arrays;
import java.util.LinkedList;

public class Allocator {

    int[][] processorLinks;
    int[] processorIds;
    int[] processorWeights;
    int[] processorPriority;
    int TACT_QUANTITY = 10000;

    int[][] taskLinks;
    int[] taskIds;
    int[] taskWeights;
    int[] taskPriority;

    int[] taskOrder;
    boolean isNextProcessorMethodEquals2;
    int[][] processingJournal;
    int[][][] sendingJournal;
    int processorQuantity;
    int taskQuantity;
```

```
//int[][] copyProcessingJournal;
int[][] copySendingJournal;

LinkedList<Integer> planningOrderId;
LinkedList<Integer> planningOrderTime;
LinkedList<Integer> processingOrderId;
LinkedList<Integer> processingOrderTime;

boolean[] isPlannedTaskArray;
int[] taskAncestorArray;
int[] taskProcessedAncestorArray;
int[] taskAncestorTimeArray;
int[] taskFinishTime;
int[] taskProcessor;

int tactCounter;
int maxCounter;
int processedTasksCounter;

boolean withoutEarlyPlanningFlag=false;

public Allocator(TaskModel taskModel, TopologyModel topologyModel, int[] taskOrder, boolean isNextProcessorMethodEquals2) {

    openModels(taskModel, topologyModel);
    this.taskOrder = taskOrder;
    this.isNextProcessorMethodEquals2 = isNextProcessorMethodEquals2;


    taskQuantity = taskModel.getLinks().length;
    processorQuantity = topologyModel.getLinks().length;
    processingJournal = new int[processorQuantity][TACT_QUANTITY];
    sendingJournal = new int[processorQuantity][TACT_QUANTITY][7];

    for (int i = 0; i < processorQuantity; i++) {
        Arrays.fill(processingJournal[i], -1);
        for (int j = 0; j < TACT_QUANTITY; j++) {
            Arrays.fill(sendingJournal[i][j], -1);
        }
    }

    planningOrderId = new LinkedList<>();
    planningOrderTime = new LinkedList<>();
    processingOrderId = new LinkedList<>();
    processingOrderTime = new LinkedList<>();

    isPlannedTaskArray = new boolean[taskQuantity];
    taskAncestorArray = new int[taskQuantity];
    taskProcessedAncestorArray = new int[taskQuantity];
    taskAncestorTimeArray = new int[taskQuantity];
    taskFinishTime = new int[taskQuantity];
    taskProcessor = new int[taskQuantity];

    for (int i = 0; i < taskQuantity; i++) {
        isPlannedTaskArray[i] = false;
        taskAncestorArray[i] = 0;
        taskProcessedAncestorArray[i] = 0;
        taskAncestorTimeArray[i] = -1;

    }


}

public void setWithoutEarlyPlanningFlag(boolean withoutEarlyPlanningFlag) {
    this.withoutEarlyPlanningFlag = withoutEarlyPlanningFlag;
}

void process() {
    makeFirstAllocation();
    performIterations();
    printModellingResult(maxCounter+1);
    //System.out.println("s");
}

int getModellingTime() {
    return maxCounter;
}

void openModels(TaskModel taskModel, TopologyModel topologyModel) {
    processorLinks = topologyModel.getLinks();
    processorIds = new int[processorLinks.length];
    processorWeights = new int[processorLinks.length];
    for (int i = 0; i < processorIds.length; i++) {
        processorIds[i] = topologyModel.getTasks().get(i).getId();
        processorWeights[i] = topologyModel.getTasks().get(i).getWeight();
    }

    taskLinks = taskModel.getLinks();
    taskIds = new int[taskLinks.length];
```

```
    taskWeights = new int[taskLinks.length];
    for (int i = 0; i < taskIds.length; i++) {
        taskIds[i] = taskModel.getTasks().get(i).getId();
        taskWeights[i] = taskModel.getTasks().get(i).getWeight();
    }
}

void makeFirstAllocation() {

    calculateTaskAncestorQuantity();
    int[] processorsIndexes = findFirstProcessorsOrder();
    setProcessorsPriority(processorsIndexes);

    for (int i = 0; i < taskQuantity; i++) {
        if (taskAncestorArray[i] == 0) {
            planningOrderId.add(i);
        }
    }

    int[] tasksIndexes = new int[planningOrderId.size()];
    for (int i = 0; i < tasksIndexes.length; i++) {
        tasksIndexes[i] = planningOrderId.get(i);
    }
    planningOrderId.clear();
    setTasksPriority();
    findFirstTasksOrder(tasksIndexes);
    boolean isTaskArrayLongerTopologyArray = false;
    if (tasksIndexes.length <= processorsIndexes.length)
        isTaskArrayLongerTopologyArray = true;
    if (isTaskArrayLongerTopologyArray) {
        for (int i = 0; i < tasksIndexes.length; i++) {
            putTaskToProcessor(tasksIndexes[i], processorsIndexes[i], 0);
        }
    } else {
        for (int i = 0; i < processorsIndexes.length; i++) {
            putTaskToProcessor(tasksIndexes[i], processorsIndexes[i], 0);
        }
        for (int i = processorsIndexes.length; i < tasksIndexes.length; i++) {
            putTaskToPlanningOrder(tasksIndexes[i], 0);
        }

    }

}

void calculateTaskAncestorQuantity() {
    for (int i = 0; i < taskQuantity; i++) {
        for (int j = 0; j < taskQuantity; j++) {
            if (taskLinks[i][j] != 0) {
                taskAncestorArray[j]++;
            }
        }
    }
}


int[] findFirstProcessorsOrder() {
    int[] processorsTime = new int[processorQuantity];
    for (int i = 0; i < processorQuantity; i++) {
        processorsTime[i] = getProcessorAverageTime(i);
    }
    int[] processorsIndexes = generateArray(processorQuantity);
    sortByIncreasing(processorsIndexes, processorsTime);


    return processorsIndexes;


}

void setTasksPriority() {
    taskPriority = new int[taskQuantity];
    for (int i = 0; i < taskQuantity; i++) {
        taskPriority[taskOrder[i]] = taskQuantity - i;
    }

}

void setProcessorsPriority(int[] processorOrder) {
    processorPriority = new int[processorQuantity];
    for (int i = 0; i < processorQuantity; i++) {
        processorPriority[processorOrder[i]] = processorQuantity - i;
    }
}

void findFirstTasksOrder(int[] tasksIndexes) {
    int[] priorities = new int[tasksIndexes.length];
    for (int i = 0; i < priorities.length; i++) {
        priorities[i] = taskPriority[tasksIndexes[i]];
```

```
    }
    sortByDecreasing(tasksIndexes, priorities);
}

void performIterations() {
    tactCounter = 0;
    processedTasksCounter = 0;
    while (processedTasksCounter != taskQuantity) {
        boolean flag = true;
        if (isNextProcessorMethodEquals2)
            flag = findNextProcessor2() > -1;

        while (!planningOrderId.isEmpty() && planningOrderTime.getFirst() <= tactCounter && flag) {
            int nextIndex = findTaskInOrder(tactCounter);
            allocateTask(nextIndex);
            if (isNextProcessorMethodEquals2)
                flag = findNextProcessor2() > -1
        }
        while (!processingOrderId.isEmpty() && processingOrderTime.getFirst() <= tactCounter) {
            releaseTask((processingOrderId.getFirst()));
            processingOrderTime.removeFirst();
            processingOrderId.removeFirst();
        }
        tactCounter++;
    }
}

void allocateTask(int taskId) {
    int nextProcessorIndex = findNextProcessor(taskId);

    int routingResultTime = route(taskId, nextProcessorIndex, true);
    int startTime = findWindowForProcessor(processingJournal[nextProcessorIndex], taskWeights[taskId], routingResultTime);
    putTaskToProcessor(taskId, nextProcessorIndex, startTime);
}

void releaseTask(int taskId) {
    processedTasksCounter++;
    for (int i = 0; i < taskQuantity; i++) {
        if (taskLinks[taskId][i] != 0) {
            taskProcessedAncestorArray[i]++;
            if (taskProcessedAncestorArray[i] == taskAncestorArray[i]) {
                putTaskToPlanningOrder(i, tactCounter + 1);
            }
        }
    }
}

int route(int toTaskIndex, int toProcessorIndex, boolean isReal) {
    int routingTime = 0;
    LinkedList<Integer> parentTasksList = new LinkedList<>();
    LinkedList<Integer> parentTasksTimeList = new LinkedList<>();
    int maxFinishTime = 0;
    if (withoutEarlyPlanningFlag) {
        for (int i = 0; i < taskQuantity; i++) {
            if (taskLinks[i][toTaskIndex] != 0) {
                if (maxFinishTime < taskFinishTime[i])
                    maxFinishTime = taskFinishTime[i];
            }
        }
    }
    for (int i = 0; i < taskQuantity; i++) {
        if (taskLinks[i][toTaskIndex] != 0) {

            parentTasksList.add(i);
            if (withoutEarlyPlanningFlag) {
                parentTasksTimeList.add(maxFinishTime);//next tact allow to start sending
            } else {
                parentTasksTimeList.add(taskFinishTime[i]);//next tact allow to start sending
            }

        }
    }
    if (!parentTasksList.isEmpty()) {

        makeCopyJournals();
        int[] parentTasksArray = listToArray(parentTasksList);
        int[] parentTasksTimeArray = listToArray(parentTasksTimeList);
        sortByIncreasing(parentTasksArray, parentTasksTimeArray);

        for (int i = 0; i < parentTasksArray.length; i++) {
            int parentTaskIndex = parentTasksArray[i];
            int weight = taskLinks[parentTaskIndex][toTaskIndex];
            int fromProcessorIndex = taskProcessor[parentTaskIndex];
            int startSendingTime = taskFinishTime[parentTaskIndex];
            if (withoutEarlyPlanningFlag) {
                startSendingTime = maxFinishTime;
            }
            if (toProcessorIndex != fromProcessorIndex) {
                int[] destinationTimeArray = new int[processorQuantity];
```

```
            int[] destinationSourceArray = new int[processorQuantity];
            Arrays.fill(destinationTimeArray, Integer.MAX_VALUE);
            Arrays.fill(destinationSourceArray, Integer.MAX_VALUE);
            destinationTimeArray[fromProcessorIndex] = startSendingTime;
            destinationSourceArray[fromProcessorIndex] = -1;
            LinkedList<Integer> wave = new LinkedList<>();
            wave.add(fromProcessorIndex);
            while (!wave.isEmpty()) {
               int currentId = wave.getFirst();
               for (int j = 0; j < processorQuantity; j++) {

                  if (processorLinks[j][currentId] != 0 || processorLinks[currentId][j] != 0) {
                     int comingTime = findWindowForLink(copySendingJournal[currentId], copySendingJournal[j], weight,
                           destinationTimeArray[currentId] + 1) + weight - 1;

                     if (destinationTimeArray[j] > comingTime) {
                        destinationTimeArray[j] = comingTime;
                        destinationSourceArray[j] = currentId;
                        wave.add(j);
                     }
                  }

               }
               wave.removeFirst();
            }


            if (routingTime < destinationTimeArray[toProcessorIndex] + 1)
               routingTime = destinationTimeArray[toProcessorIndex] + 1;

            if (isReal) {
               int currentIndex = toProcessorIndex;
               LinkedList<Integer> processorIndexes = new LinkedList<>();
               LinkedList<Integer> processorStartSendingTimes = new LinkedList<>();
               while (currentIndex != fromProcessorIndex) {
                  processorIndexes.add(currentIndex);
                  processorStartSendingTimes.add(destinationTimeArray[currentIndex] - weight + 1);
                  currentIndex = destinationSourceArray[currentIndex];
               }
               processorIndexes.add(fromProcessorIndex);

               for (int j = processorStartSendingTimes.size() - 1; j >= 0; j--) {
                  int tempFromProcessorIndex = processorIndexes.get(j + 1);
                  int tempToProcessorIndex = processorIndexes.get(j);
                  int startTime = processorStartSendingTimes.get(j);
                  int[] parametresFrom = {parentTaskIndex, toTaskIndex, fromProcessorIndex, toProcessorIndex,
                        tempFromProcessorIndex, tempToProcessorIndex, 0};
                  int[] parametresTo = {parentTaskIndex, toTaskIndex, fromProcessorIndex, toProcessorIndex,
                        tempFromProcessorIndex, tempToProcessorIndex, 1};

                  for (int k = 0; k < weight; k++) {
                     int tempTact = k + startTime;
                     copySendingJournal[tempFromProcessorIndex][tempTact] = 1;
                     copySendingJournal[tempToProcessorIndex][tempTact] = 1;
                     System.arraycopy(parametresFrom, 0, sendingJournal[tempFromProcessorIndex][tempTact], 0, 7);
                     System.arraycopy(parametresTo, 0, sendingJournal[tempToProcessorIndex][tempTact], 0, 7);
                  }
               }
            }
         }
      }
   }
   return routingTime;
}

int findTaskInOrder(int time) {
   int result = -1;
   LinkedList<Integer> tempList = new LinkedList<>();
   for (int i = 0; i < planningOrderId.size(); i++) {
      if (planningOrderTime.get(i) <= time) {
         tempList.add(planningOrderId.get(i));

      } else {
         break;
      }
   }

   int[] indexes = listToArray(tempList);
   int[] priorities = new int[indexes.length];
   for (int i = 0; i < priorities.length; i++) {
      priorities[i] = taskPriority[indexes[i]];
   }

   sortByDecreasing(indexes, priorities);

   result = indexes[0];

   for (int i = 0; i < planningOrderId.size(); i++) {
      if (planningOrderId.get(i) == result) {
```

```
            planningOrderId.remove(i);
            planningOrderTime.remove(i);
        }
    }

    return result;
}

int findNextProcessor(int taskId) {

    int result = 0;
    if (isNextProcessorMethodEquals2) {
        result = findNextProcessor2();
    } else {
        result = findNextProcessor7(taskId);
    }
    return result;
}


int findNextProcessor2() {
    LinkedList<Integer> tempProcessorIndexesList = new LinkedList<>();
    LinkedList<Integer> tempProcessorTimesList = new LinkedList<>();
    for (int i = 0; i < processorQuantity; i++) {
        if (processingJournal[i][tactCounter] == -1) {
            int time = 0;
            for (int j = tactCounter - 1; j >= 0; j--) {
                if (processingJournal[i][j] == -1) {
                    time++;
                } else {
                    break;
                }
            }
            if (time >= 0) {
                boolean flag = true;
                for (int j = tactCounter; j < processingJournal[i].length && flag; j++) {
                    if (processingJournal[i][j] != -1)
                        flag = false;
                }
                if (flag) {
                    tempProcessorIndexesList.add(i);
                    tempProcessorTimesList.add(time);
                }

            }

        }
    }

    if (tempProcessorIndexesList.size() > 0) {
        int[] tempProcessorIndexesArray = listToArray(tempProcessorIndexesList);

        int[] tempProcessorTimesArray = listToArray(tempProcessorTimesList);
        sortByDecreasing(tempProcessorIndexesArray, tempProcessorTimesArray);
        return tempProcessorIndexesArray[0];
    } else return -1;

}

int findNextProcessor7(int taskId) {

    int[] processorTimes = new int[processorQuantity];
    int minIndex = 0;
    for (int i = 0; i < processorQuantity; i++) {
        int routingResult = route(taskId, i, false);
        processorTimes[i] =
                findWindowForProcessor(processingJournal[i], taskWeights[taskId], routingResult);
        if (processorTimes[i] < processorTimes[minIndex])
            minIndex = i;
    }
    return minIndex;
}

int findWindowForProcessor(int[] array, int weight, int time) {
    int currentWindow = 0;
    int result = time;
    boolean flag = true;
    while (flag) {

        if (array[result] == -1) {
            currentWindow++;
        } else {
            currentWindow = 0;
        }
        if (currentWindow == weight) {
            flag = false;
        } else {
            result++;
        }
```

```
        }

        return result - weight + 1;

    }

    int findWindowForLink(int[] array1, int[] array2, int weight, int time) {
        int currentWindow = 0;
        int result = time;
        boolean flag = true;
        while (flag) {

            if (array1[result] == -1 && array2[result] == -1) {
                currentWindow++;
            } else {
                currentWindow = 0;
            }

            if (currentWindow == weight) {
                flag = false;
            } else {
                result++;
}
        }

        return result - weight + 1;

    }

    int findWindowForLink(int[] array1, int[] array2, int weight, int time) {
        int currentWindow = 0;
        int result = time;
        boolean flag = true;
        while (flag) {

            if (array1[result] == -1 && array2[result] == -1) {
                currentWindow++;
            } else {
                currentWindow = 0;
            }

            if (currentWindow == weight) {
                flag = false;
            } else {
                result++;
            }
        }

        return result - weight + 1;
    }

    int[] listToArray(LinkedList<Integer> list) {
        int[] array = new int[list.size()];
        for (int i = 0; i < list.size(); i++) {
            array[i] = list.get(i);
        }
        return array;
    }

    void putTaskToProcessor(int taskId, int processorId, int tact) {
        int finishTime = tact + taskWeights[taskId] - 1;
        int place = findPlaceByTime(processingOrderTime, finishTime);
        processingOrderId.add(place, taskId);
        processingOrderTime.add(place, finishTime);
        for (int i = 0; i < taskWeights[taskId]; i++)
            processingJournal[processorId][tact + i] = taskId;
        taskFinishTime[taskId] = finishTime;
        taskProcessor[taskId] = processorId;
        if (finishTime > maxCounter)
            maxCounter = finishTime + 1;
    }

    void putTaskToPlanningOrder(int taskId, int tact) {
        int place = findPlaceByTime(planningOrderTime, tact);
        planningOrderId.add(place, taskId);
        planningOrderTime.add(place, tact);
    }

    int findPlaceByTime(LinkedList<Integer> taskTimes, int tact) {
        int result = 0;
        switch (taskTimes.size()) {
            case 0: {
                return 0;
            }
            case 1: {
                if (tact < taskTimes.getFirst())
                    return 0;
                else
                    return 1;
```

```
        }
        default:
            for (int i = 0; i < taskTimes.size(); i++) {
                if (taskTimes.get(i) < tact)
                    result++;
                else
                    break;
            }
            break;
    }
    return result;
}

boolean isFreeProcessorHere(int tact) {
    int count = getFreeProcessorCount(tact);
    if (count > 0)
        return true;
    else
        return false;
}

int getFreeProcessorCount(int tact) {
    int result = 0;
    for (int i = 0; i < processorQuantity; i++) {
        if (processingJournal[i][tact] == -1) {
            boolean flag = true;
            for (int j = tact; j < tact + 100 && flag; j++) {
                if (processingJournal[i][j] != -1)
                    flag = false;
            }
            if (flag)
                result++;
        }

    }
    return result;
}


void makeTempJournals(/*int[][] array1, */ int[][] array) {
     for (int i = 0; i < processorQuantity; i++) {
        for (int j = 0; j < TACT_QUANTITY; j++) {
            //array1[i][j]=processingJournal[i][j];
            if (sendingJournal[i][j][0] != -1) {
                array[i][j] = 1;
            } else {
                array[i][j] = -1;
            }
        }
    }
}

void makeCopyJournals() {
    copySendingJournal = new int[processorQuantity][TACT_QUANTITY];
    makeTempJournals(/*copyProcessingJournal, */copySendingJournal);
}

int[] generateArray(int n) {
    int[] result = new int[n];
    for (int i = 0; i < n; i++) {
        result[i] = i;
    }
    return result;
}

void sortByDecreasing(int[] ids, int[] weight) {
    for (int i = 0; i < ids.length; i++) {
        for (int j = i + 1; j < ids.length; j++) {
            if (weight[i] < weight[j]) {
                int bufId = ids[i];
                int bufWeight = weight[i];
                ids[i] = ids[j];
                weight[i] = weight[j];
                ids[j] = bufId;
                weight[j] = bufWeight;
            }
        }
    }
}

void sortByIncreasing(int[] ids, int[] weight) {
    for (int i = 0; i < ids.length; i++) {
        for (int j = i + 1; j < ids.length; j++) {
            if (weight[i] > weight[j]) {
                int bufId = ids[i];
                int bufWeight = weight[i];
                ids[i] = ids[j];
                weight[i] = weight[j];
                ids[j] = bufId;
```

```
                weight[j] = bufWeight;
            }
        }
    }
}


int getProcessorAverageTime(int start) {
    int[] destinationTime = new int[processorQuantity];
    Arrays.fill(destinationTime, Integer.MAX_VALUE);
    destinationTime[start] = 0;
    LinkedList<Integer> wave = new LinkedList<>();
    wave.add(start);
    while (!wave.isEmpty()) {
        int currentId = wave.getFirst();
        for (int i = 0; i < processorQuantity; i++) {

            if (processorLinks[i][currentId] != 0 || processorLinks[currentId][i] != 0) {
                if (destinationTime[i] > destinationTime[currentId] + 1) {
                    destinationTime[i] = destinationTime[currentId] + 1;
                    wave.add(i);
                }
            }

        }
        wave.removeFirst();
    }
    int sum = 0;
    for (int i = 0; i < processorQuantity; i++)
        sum += destinationTime[i];
    return sum;
}

void printTranslated(int[] arrayFrom, int[] arrayTo) {
    if (arrayFrom.length > 0) {
        String result = "[";
        for (int i = 0; i < arrayFrom.length - 1; i++) {
            result += arrayTo[arrayFrom[i]] + ", ";
        }
        result += arrayTo[arrayFrom[arrayFrom.length - 1]] + "]";
        System.out.println(result);
    } else {
        System.out.println("[]");
    }
}

void printModellingResult(int tacts) {
    String title = makeSpace(8);
    for (int j = 0; j < tacts; j++) {
        title += " " + makeSpace(7) + intToString(j + 1);
    }
    System.out.println(title);
    for (int i = 0; i < processorQuantity; i++) {
        String lineProcessor = "PµP " + intToString(processorIds[i]) + " pr:";
        String lineTransmition = "PµP " + intToString(processorIds[i]) + " io:";
        for (int j = 0; j < tacts; j++) {

            int taskId = processingJournal[i][j];
            String taskStr = "";
            if (taskId != -1) {
                taskStr = intToString(taskIds[taskId]);
            } else {
                taskStr = "  ";
            }
            lineProcessor += " " + makeSpace(7) + taskStr;

            int taskFrom = sendingJournal[i][j][0];
            String transmPart = "";
            if (taskFrom != -1) {
                int direction = sendingJournal[i][j][6];
                if(direction!=1) {
                    taskFrom = taskIds[taskFrom];
                    int taskTo = taskIds[sendingJournal[i][j][1]];

                    int processorId = 0;
                    processorId = processorIds[sendingJournal[i][j][5]];

                    transmPart = intToString(taskFrom) + "-" + intToString(taskTo) + "(" + intToString(processorId) + ")";
                }

                else {
                    transmPart = makeSpace(9);
                }

            } else {
                transmPart = makeSpace(9);
            }
            lineTransmition += " " + transmPart;
        }
```

```
            System.out.println(lineProcessor);
            System.out.println(lineTransmition);
            System.out.println("");
        }

    }

    String makeSpace(int n) {
        String result = "";
        for (int i = 0; i < n; i++)
            result += " ";
        return result;
    }

    String intToString(int n) {
        String result = "";
        if (n > 0 && n < 10)
            result = " ";
        result += n;
        return result;
    }
}
```